ESPRIT III

ORES: TOWARDS THE FIRST GENERATION OF TEMPORAL DBMS (P7224)

DELIVERABLE D2

SPECIFICATION OF VALID TIME SQL

Responsible: 01 PLIROFORIKI

APRIL 1993

SPECIFICATION OF VALID TIME SQL (DELIVERABLE D2)

Responsible: 01 PLIROFORIKI

ABSTRACT

This report concerns the specification of a valid time extension to SQL. It is a consistent extension in that it preserves the syntax and semantics of SQL. In addition, it incorporates all the operations of Valid Time Relational Algebra.

TABLE OF CONTENTS

1. INTRODUCTION

In this report we formalise a consistent extension to SQL, namely Valid Time SQL (VT-SQL), which can handle both snapshot and valid time data. VT-SQL can be seen as an integration of SQL and Valid Time Relational Algebra (VT-SQL) [01P 93], in the sense that it preserves the syntax and semantics of the former and extends them in a natural way so as to include all the characteristics of the latter. As a consequence, VT-SQL is as user-friendly as SQL. The new features are the following:

- Predefined constants
- New data types
- New literals
- New relational operators
- New scalar functions
- New aggregate functions
- A consistent extension to the SQL CREATE TABLE statement
- New clauses for the incorporation of the VT-RA **reformat** and **normalise** operations
- A consistent extension to the SQL UNION operation
- A direct incorporation of the **except** and **pexcept** operations of VT-RA
- A consistent extension to the SQL data manipulation statements

Effort has been made in defining an almost full SQL extension and not a minimal one, as specified in the Technical Annex of the ORES project. The specification of VT-SQL has been based on the following:

- Literature on standard SQL, especially [Date 86], [Lans 88a], [Lans 88b] and [Ingres 89]
- Literature on the definition of a VT-SQL ([Navathe & Ahmed 86], [Sarda 90])
- The user requirements, as specified in [CPH 93].

It is worth mentioning that VT-SQL is so powerful that it can answer more complicated queries than those identified in the test bed application [CPH 93].

Report C3 [01P 93] is a requirement for the understanding of this one. The remainder of this report is outlined as follows.

In section 2 we provide a sample database, against which we provide examples on the definition of VT-SQL. In section 3 we describe VT-SQL. In section 4 we justify certain specification decisions. In section 5 we present implementation problems and identify the portion of VT-SQL which will be implemented within the ORES project. Conclusions are drawn in the last section. The document is followed by three appendices. In appendix A we provide certain formal definitions which are necessary to VT-SQL. In appendix B we provide the full VT-SQL syntax. In appendix C we demonstrate how VT-SQL can be used to answer queries of the test bed application. All the examples provided in this appendix answer queries which are real in nature, not hypothetical.

2. A SAMPLE DATABASE

In this section we present a sample database which is used in the examples provided in subsequent sections.

Since a Valid Time DBMS (VT-DBMS) should support various time-interval types, in the following we usually refer to generic *time-points* (d5, d6, d7 etc.) and *time-intervals* ([d5, d7), [d12, d15) etc.). A generic time-interval over an arbitrary set of time-points is alternatively denoted by δx, δy or δz. A *time* type is either a *time-point* or *time-interval* type.

Although dates and intervals over a set of dates are actually displayed in a format like 30/01/93 and [30/01/93, 20/04/93), for simplicity reasons we use the above notation and denote them like d5 and [d5, d10), respectively.

Hour-intervals are intervals over the set HOURS = {h1, h2, ..., h25} (for example [h14, h25)).

Month-intervals are intervals over the set MONTHS = {m1, m2, ..., m13} (for example [m7, m13)).

Tables

For each of the tables which follow we provide its key and a short description of its contents.

SALARY

Name	Amount	Time
John	10K	[d2, d6)
John	10K	[d9, d12)
John	12K	[d15, d18)
Alex	14K	[d9, d12)

Key: <*Name-i, Time-p*>

The salary of each employee for each of the dates in the Time interval.

ASSIGNMENT

Name	Dept.	Time
John	shoe	[d3, d7)
John	food	[d7, d11)
John	toys	[d11, d15)
Alex	shoe	[d5, d10)
Mary	toys	[d5, d11)

Key: <*Name-i, Time-p*>

The department in which each employee was assigned for each of the dates in the Time interval.

PROJECT

Name	Project	Time
John	P1	[d1, d5)
John	P2	[d2, d12)
John	P1	[d15, d30)
Mary	P1	[d2, d10)

Key: <*Name-i, Project-i, Time-p*>

The project in which each employee was involved for each of the dates in the Time interval.

SHIFT

Name	Day	Hour
John	[d1, d10)	[h1, h9)
John	[d5, d20)	[h9, h12)

Key: <*Day-p, Hour-p*>

The interval of hours each employee works, for each particular interval of dates.

INFLATION

Country	Percentage	Time
A	8.0	[m1, m4)
A	10.0	[m4, m7)
A	10.0	[m1, m13)

Key: <*Country-i, Time-i*>

The rate at which the inflation of each country was running, during a particular month-interval.

SALES

Date	Product	Qty	Price
d5	apples	500	100
d5	grapes	200	300
d5	oranges	300	150
d5	pears	400	200
d6	apples	600	120
d6	grapes	250	300
d6	oranges	350	170
d6	pears	200	220
d7	apples	650	120
d7	grapes	300	300
d8	oranges	400	170
d8	pears	300	220

Key: <*Date-i, Product-i*>

The quantity sold and the unit price of each product on a particular date.

3. *EXTENSIONS TO SQL*

In this section we give the specification of VT-SQL. The new features are given in bold, to make reading easier. For each new feature we provide a description, followed by appropriate examples and its syntax in BNF. Similarly to [Lans 88b] we have avoided complicating the syntax, by writing under appropriate headings the rules which cannot be deduced by it. A complete syntax of VT-SQL is given in appendix B.

3.1 Predefined Constants

Description

If D={d_1, d_2, ..., d_n} is a set of consecutive time-points, then two predefined constants, *minD* and *maxD*, equal d_1 and d_n, respectively. If *minD* or *maxD* are in an expression, the VT-DBMS replaces them by their values before the expression is evaluated.

Two predefined constants of particular interest are *mindate* and *maxdate* which equal the least and greatest date supported by a VT-DBMS. The importance of these constants is shown in subsequent sections.

3.2 New Data Types

Description

For every set D={d_1, d_2, ..., d_n} of consecutive time-points, DINTERVAL is a new data type with elements of the form [d_i, d_j), where $d_i < d_j$.

A data type of particular interest is DATEINTERVAL. For simplicity reasons, we again use the notation [d_i, d_j), for elements of a DATEINTERVAL type, where d_i, d_j are of type DATE.

Example:

```
CREATE TABLE SALARY (Name  VARCHAR(15) NOT NULL,
        Amount INTEGER4,
        Time  DATEINTERVAL NOT NULL)
```

Format

<data-type> ::= <standard-SQL-data-type>
| DATEINTERVAL

General Rules

1. Every SQL data type is also valid in VT-SQL.
2. The new data types may be used in exactly the same places where the SQL data types are used.
3. The rules to which the SQL data types obey, are exactly the same for the VT-SQL types.

Convention

A time-interval which participates in the operations defined in the sub-sections which follow, may not be *semi-null* or null. Examples of semi-null time intervals are '[, d10)' and '[d5,)'.

3.3 New Literals

Description

If 'd_i' and 'd_j' are time-point literals, $d_i < d_j$, then '[d_i, d_j)' is a DINTERVAL literal.

Example:

'[01/01/93, 25/03/93)' is a DATEINTERVAL literal.

```
INSERT INTO INFLATION(Country, Percentage, Time)
VALUES ('B', 3.0, '[m7, m13)')
```

Format

<literal> ::= <standard-SQL-literal>
| <dateinterval-literal>

<dateinterval-literal> ::= '[<date>, <date>)'

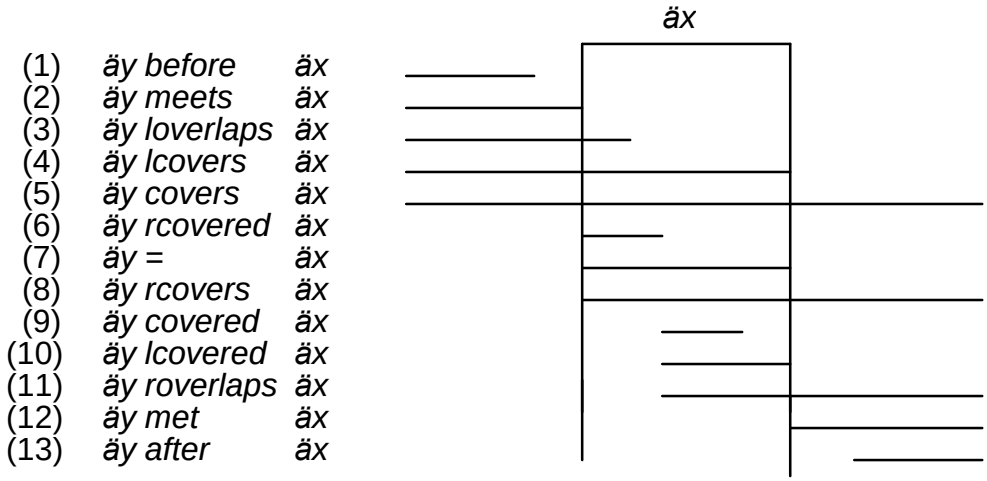
General Rules

1. Every SQL literal is also valid in VT-SQL.
2. The new literals may be used in exactly the same places where the SQL literals are used.
3. The rules to which the SQL literals obey, are exactly the same for the VT-SQL literals.

3.4 New Relational Operators

Description

The time-interval relational operators are shown in figure 1. Their definition is given in appendix A.



dy psubinterv dx Û (6 V 9 V 10) (dy is a pure subinterval of dx)

dy subinterv dx Û (6 V 9 V10 V 7) (dy is a subinterval of dx)

dy psupinterv dx Û (4 V 5 V 8) (dy is a pure superinterval of dx)

dy supinterv dx Û (4 V 5 V 8V7) (dy is a superinterval of dx)

dy overlaps dx Û (3 V 11)

dy merges dx Û (2 V 3 V . . . V12)

dy cp dx Û (3 V 4 V . . . V 11) (dy has common points with dx)

dy precedes dx Û (1 V 2 V 3 V 4 V 5 V 6)

dy follows dx Û (8 V9 V 10 V11 V 12 V 13)

dy prequals dx Û (1 V 2 V 3 V 4 V5 V 6 V 7) (dy precedes or equals dx)

dy folequals dx Û (7 V 8 V9 V 10 V 11V 12 V13) (dy follows or equals dx)

dy adjacent dx Û (2 V12)

Figure 1: Internal relational operations.

Example:

[d12, d50) *before* [d60, d80) is true.
[d12, d60) *meets* [d60, d80) is true.
[d12, d60) *meets* [d70, d80) is false.
[d12, d50) *cp* [d30, d80) is true.
[d12, d50) *cp* [d60, d80) is false.

"*Give the name and the department of the employees who worked at some time in* [d5, d10)."

```
SELECT   Name, Dept
FROM     ASSIGNMENT
WHERE    Time cp '[d5,  d10)'
```

"*Give the employees who were not paid at any day in [d5, d10)*."

```
SELECT   Name
FROM     SALARY
WHERE    Name not in   (SELECT     Name
                        FROM       SALARY S1
                        WHERE      S1.Name cp '[d5, d10)'
                        GROUP BY   Name          )
GROUP BY Name
```

"*Give the employees whose salary exceeded 9K at all days in [d5, d10)*."

```
SELECT   Name
FROM     SALARY
WHERE    Time cp '[d5, d10)'
AND      Name not in   (SELECT     Name
                        FROM       SALARY S1
                        WHERE      S1.Time cp '[d5, d10)'
                        AND        S1.Amount <= 9K
                        GROUP BY   Name)
GROUP BY Name
```

"*Give the employees who left the toys department at least one day before John moved to it*." (We assume that John moved to the toys department for the first time.)

```
SELECT   Name
FROM     ASSIGNMENT
WHERE    Dept = 'toys'
AND      Time before  (SELECT   Time
                      FROM     ASSIGNMENT A
                      WHERE    A.Name = 'John'
                      AND      A.Dept = 'toys')
GROUP BY Name
```

Format

<comparison-op>::= = | <> | < | > | <= | >=
| <interval-comparison-op>

<interval-comparison-op> ::= *before*
| *meets*
| *loverlaps*
| *lcovers*
| *covers*
| *rcovered*
| *=*
| *rcovers*
| *covered*
| *lcovered*
| *roverlaps*
| *met*
| *after*
| *psubinterv*
| *subinterv*
| *psupinterv*
| *supinterv*
| *overlaps*
| *merges*
| *cp*
| *precedes*
| *follows*
| *prequals*
| *folequals*
| *adjacent*

General Rules

1. Every SQL literal is also valid in VT-SQL.
2. The new relational operators may be used in exactly the same places where the SQL relational operators are used.
3. The rules to which the SQL relational operators obey, are exactly the same for the VT-SQL relational operators.

Additional Rule

The new operators may be used only between time-intervals of the same type.

3.5 New Scalar Functions

Description

The scalar functions which can be used in VT-SQL are those defined in [01P 93]. Their definition is given in appendix A.

Examples

now : It returns the current time.

start([d5, d8)) = d5

stop([d5, d8)) = d8

topoint([d5, d6) = d5

tointerv(d5) = [d5, d6)

interv(d5, d8) = [d5, d8)
interv(d5, d3) = *undefined*

intervsect([d5, d10), [d7, d12)) = [d7, d10)
intervsect([d5, d10), [d15, d20)) = *undefined*

succ(d5, 3) = d8
succ(d5, -3) = d2

dur([d5, d8)) = 3

span(d3, d8) = -5

span(d8, d3) = 5

dist(d3, d8) = 5

middle([d5, d8)) = d6
middle([d5, d9)) = d6

merge([d5, d8), [d6, d12)) = [d5, d12)
merge([d5, d8), [d10, d20)) = *undefined*

window(d5, 5, 0) = [d5, d10)
window(d5, 5, 1) = [d10, d15)
window(d5, 3, -1) = [d2, d5)

windowno(d5, 5, d17) = 2
windowno(d5, 5, d15) = 2
windowno(d10, 5, d9) = -1

"*Give the names of employees whose current salary exceeds 10K*."

```
SELECT      Name
FROM        SALARY
WHERE       tointerv('now') cp Time
GROUP BY    Name
```

"*For how long have we been knowing John*?" (It is assumed that we know an employee from the first time he started working in some department.)

```
SELECT      'now' - min(start(Time))
FROM        ASSIGNMENT
WHERE       Name = 'John'
```

"*Give John's salary and assignment to departments, for all times at which both his salary and assignment to departments was known.*"

```
SELECT`  Amount, Dept,      intervsect(A.Time, S.Time)
FROM     ASSIGNMENT A,      SALARY S
WHERE    A.Name = 'John'
AND      A.Name = S.Name
AND      A.Time  cp  S.Time
```

"*Give the names of employees who moved from the shoe to the food department while Mary was assigned to some project*."

```
SELECT      A1.Name
FROM        ASSIGNMENT A1, ASSIGNMENT A2, PROJECT P
WHERE       A1.Name = A2.Name
AND         A1.Dept = 'shoe'
AND         A2.Dept = 'food'
AND         A1.Time meets A2.Time
AND         tointerv(start(A2.Time)) cp P.Time
AND         P.Name = 'Mary'
GROUP BY    Name
```

"*Give the current salary of all employees not employed at any time in [d5, d12).*"

```
SELECT    S1.Name, S1.Amount
FROM      SALARY S1
WHERE     S1.Time cp tointerv('now')
AND       Name not in   (SELECT   S2.Name
                         FROM     SALARY S2
                         WHERE    S1.Name = S2.Name
                         AND      S2.Time cp '[d5, d10)')
```

"*Give the employees whose first salary was greater than 8K.*"

```
SELECT    S1.Name
FROM      SALARY S1
WHERE     S1.Amount > 8K
AND       stop(S1.Time) <= all   (SELECT   start(S2.Time)
                                  FROM     SALARY S2
                                  WHERE    S2.Name =  S1.Name
                                  AND      S2.Time  <> S1.Time)
GROUP BY S1.Name
```

Format

```
<scalar-function> ::=     <standard-SQL-scalar-function>
                    |     now
                    |     start(Dateinterval)
                    |     stop (Dateinterval)
```

| *topoint*(Dateinterval)
| *tointerv*(Date)
| *interv*(Date, Date)
| *intervsect*(Dateinterval, Dateinterval)
| *succ*(Date, Integer)
| *dur*(Dateinterval)
| *span*(Date, Date)
| *dist*(Date, Date)
| *middle*(Dateinterval)
| *merge*(Dateinterval, Dateinterval)
| *window*(StartDate, TimeDuration, TimeNumber)
| *windowno*(StartDate, TimeDuration, Date)

General Rules

1. Every SQL scalar function is also valid in VT-SQL.
2. The new scalar functions may be used in exactly the same places where the SQL scalar functions are used.
3. The rules to which the SQL scalar functions obey, are exactly the same for the new scalar functions.

Additional Rule

The arguments of every new scalar function *funct*, must be *funct-compatible.*

3.6 New Aggregate Functions

Description

Two new aggregate functions are the following.

Function *countap* (*count all time-points*) returns the number of time-points that are contained in a selected column or value expression of a time-interval type. Duplicate points are also counted.

Function *countdp* (*count distinct time-points*) returns the number of *distinct* time-points that are contained in a selected column or value expression of a time-interval type.

Examples

"*Give the total manpower put to the shoe department, measured in days.*"

For simplicity reasons, we assume that each employee works on all the dates in a specified time-interval.

```
SELECT   countap(Time)
FROM     ASSIGNMENT
WHERE    Dept = 'toys'
```

We can see that only the first and fourth row of ASSIGNMENT satisfy the search condition. For the associated time-intervals, we notice the following:

Interval	Days in Interval	Number of Days in Interval
[d3, d7)	d3, d4, d5, d6	4
[d5, d10)	d5, d6, d7, d8, d9	5
	Total manpower in days:	9

Hence, the result obtained is 9.

"*Give the number of distinct days on which John was assigned to some project.*"

```
SELECT   countdp(Time)
FROM     PROJECT
WHERE    Name = 'John'
```

Interval	Days in Interval
[d1, d5)	d1, d2, d3, d4
[d2, d12)	d2, d3, d4, ..., d11
[d15, d30)	d15, d16, ..., d29
Total number of distinct days : 26	

Hence, the result obtained is 26.

The SQL aggregate functions may also be used in order to answer valid time queries, as is demonstrated by the following examples.

Let us assume that in column Amount of SALARY we record the daily salary of employees. We recall that SALARY is a normalised table, therefore consider the query "*give the total amount paid for salaries on date d9*".

```
SELECT    sum(Amount)
FROM      SALARY
WHERE     tointerv('d9') cp Time
```

"*Give the employees who were first employed on a date greater than d7*."

```
SELECT      Name
FROM        SALARY
GROUP BY    Name
HAVING      min(start(Time)) > 'd7'
GROUP BY    Name
```

"*Give the total amount paid for salaries from d5 to d15*."

```
SELECT    sum(Amount*dur(intervsect(Time, '[d5, d16)')))
FROM      SALARY
WHERE     tointerv('[d5, d16)') cp Time
```

"*Give the number of salary increases for John.*"

```
SELECT    count(S2.Time)
FROM      SALARY S1, SALARY S2
WHERE     S1.Name = 'John'
AND       S1.Name = S2.Name
AND       S1.Amount < S2.Amount
AND       S1.Time meets S2.Time
```

"*Give the number of employees who had a salary increase in [d5, d15).*"

```
SELECT    count(distinct S2.Name)
FROM      SALARY S1, SALARY S2
WHERE     S1.Name = S2.Name
AND       S1.Amount < S2.Amount
AND       S1.Time meets S2.Time
AND       tointerv(start(S2.Time)) cp '[d5, d15)'
```

"*Give the department in which John was working when he got his last salary increase.*"

```
SELECT   Dept
FROM     ASSIGNMENT
WHERE    Name = 'John'
AND      Time cp   (SELECT   tointerv(max(start(S2.Time)))
                   FROM     SALARY S1, SALARY S2
                   WHERE    S1.Name   =  'John'
                   AND      S1.Name   =  S2.Name
                   AND      S1.Amount <  S2.Amount
                   AND      S1.Time meets S2.Time)
```

Format

<distinct-set-function> ::= <standard-SQL-distinct-set-function>
| { COUNTDP | COUNTAP} (DISTINCT <column-spec>)

<all-set-function> ::= <standard-SQL-all-set-function>
| { COUNTDP | COUNTAP} [ALL] <value-exp>)

General Rules

1. Every SQL aggregate function is also valid in VT-SQL.
2. The new aggregate functions may be used in exactly the same places where the SQL functions are used.
3. The rules to which the SQL aggregate functions obey, are exactly the same for the new aggregate functions.

Additional Rules

1. The argument of every new aggregate function must be either a column or a value expression of a time-interval type.
2. Functions AVG, MAX, MIN and SUM may not be applied to elements of a time-interval type.

3.7 Extension to the Create Table Statement

Description

The syntax of the SQL2 CREATE TABLE command, enables the definition of the primary key of a table. Its syntax is

<table-definition> ::=
    CREATE TABLE <table name>
    (<column-name> <data-type> <other> {, <column-name> <data-type> <other> ...}
    [, PRIMARY KEY (<key-column-list>)])
<key-column-list> ::=
    <key-column-name> {, <key-column-name> ...})

where <other> denotes other SQL2 declarations which are beyond the purposes of ORES. Thus,

```
CREATE TABLE SP     (Supplier    CHAR(10),
                     Part        CHAR(10),
                     Quantity    INTEGER )
```

is an example of declaring a table. If we are also interested in declaring that <Supplier, Part> is the key of the table, the syntax is

```
CREATE TABLE SP     (Supplier        CHAR(10),
                     Part            CHAR(10),
                     Quantity        INTEGER,
                     PRIMARY KEY     (Supplier, Part))
```

Assuming that an INTEGERINTERVAL data type is also supported, another example is

```
CREATE TABLE OWNERSHIP    (Name   CHAR(10),
                           Length INTEGERINTERVAL,
                           Width  INTEGERINTERVAL,
                           PRIMARY KEY (Length, Width))
```

The constraint imposed by the primary key of OWNERSHIP implies that each piece of land (rectangle with (Length, Width) co-ordinates) can be owned by at most one person.

Data is inserted in such tables in the usual way and if the primary key has been declared then a table may not have two rows with the same key value. As we have seen however, if a database supports an

interval data type then it is often the case that the data of certain columns has to be normalised. Therefore, we extend the above syntax in VT-SQL as follows:

<table-definition> ::=

CREATE TABLE <table name>

(<column-name> <data-type> <other> {, <column-name> <data-type> <other>...}

[, NORMALISED (<normalised-column-list>)]

[, PRIMARY KEY (<key-column-list>)]))

<normalised-column> ::= <column-name>

<key-column-list> ::=

<key-column-name> [INTERVAL | POINT] {, <key-column-name> [INTERVAL | POINT] ...})

Now, whenever data is inserted, deleted or updated, <table-name> is normalised with respect to the columns whose name appears after the keyword NORMALISED. Furthermore, a <key-column-name> is followed by one of the keywords INTERVAL or POINT. From these, INTERVAL is the default and may follow a <key-column-name> only in the case that <key-column-name> does not follow the keyword "NORMALISED". If however it has been declared that a table is defined, then all the colums on which a normalisation occurs, have to participate in the key, followed by the keyword POINT. This is demonstrated by the following examples.

Examples

```
CREATE TABLE SP     (Supplier   CHAR(10),
                     Part       CHAR(10),
                     Quantity   INTEGER,
                     PRIMARY KEY (Supplier INTERVAL, Part INTERVAL))

CREATE TABLE OWNERSHIP     (Name   CHAR(10),
                            Length INTEGERINTERVAL,
                            Width  INTEGERINTERVAL,
                            PRIMARY KEY (Length INTERVAL, Width
                            INTERVAL))

CREATE TABLE INFLATION     (Country     CHAR(15),
                            Percentage  REAL,
                            Time        MONTHINTERVAL,
                            PRIMARY KEY (Country INTERVAL, Time
                            INTERVAL))
```

The above examples show how the syntax is extended if no normalisation is required. The keyword INTERVAL may be omitted from any place.

```
CREATE TABLE SALARY     (Name     CHAR(15),
                         Time     DATEINTERVAL,
                         Amount   INTEGER,
                         NORMALISED (Time))
```

The consequence of this syntax is that if we issue a command to insert the row

(John, 10K, [d6, d9))

then, after insertion, SALARY will consist of the tuples

(John, 10K, [d2, d12))
(Alex, 14K, [d9, d12))

Alternatively, if we issue a command to insert the row

(John, 11K, [d5, d8))

then, after insertion, SALARY will consist of the tuples

(John, 10K, [d2, d6))
(John, 11K, [d5, d8))
(John, 10K, [d9, d12))
(John, 12K, [d15, d18))
(Alex, 14K, [d9, d12))

We now notice that for date d5, John has two distinct salaries, 10k and 11k. This is because no primary key has been declared. To avoid therefore the insertion of two distinct salaries, SALARY has to be declared as

```
CREATE TABLE SALARY     (Name     CHAR(15),
                         Time     DATEINTERVAL,
                         Amount   INTEGER,
                         NORMALISED (Time),
                         PRIMARY KEY (Name INTERVAL, Time POINT))

CREATE TABLE SHIFT      (Name     CHAR(15),
                         Date     DATEINTERVAL,
                         Hour     INTEGERINTERVAL,
                         NORMALISED (Date, Hour),
                         PRIMARY KEY (Date POINT, Hour POINT))
```

The primary key declaration implies that no two employees may be working at the same date and hour. Furthermore, NORMALISED (Date, Hour) denotes that after insertions, deletions and updates, a normalisation always takes place firstly on Date and then on Hour.

Details concerning how data is inserted, deleted or updated in normalised tables, is given in section 3.11 which follows.

Format

<table-definition> ::=

CREATE TABLE <table name>
(<column-name> <data-type> <other> {, <column-name> <data-type> <other> ...}
[, NORMALISED (<normalised-column-list>)]
[, PRIMARY KEY (<key-column-list>)])

<normalised-column-list> ::= <column-name>

<key-column-list> ::=

<key-column-name> [INTERVAL | POINT] {, <key-column-name> [INTERVAL | POINT] ...})

General Rules

1. Each <key-column-name> in <key-column-list> must be a <column-name>.
2. A <key-column-name> may not appear more than once in <key-column-list>.
3. <other> refers to other SQL2 declarations. They will be supported in ORES if they are directly supported by INGRES.

Additional Rules

1. Each <normalised-column-name> in <normalised-column-list> must be a <column-name>.
2. A <normalised-column-name> may not appear more than once in <normalised-column-list>.
3. If NORMALISED (<normalised-column-list>) has been declared then the table is always normalised with respect to the columns in <normalised-column-list> in both data insertion and deletion and update. The <normalised-column-list> determines the sequence in which this normalisation takes place.
4. If both NORMALISED (<normalised-column-list>) and PRIMARY KEY (<key-column-list>) have been declared then every <mormalised-comlumn-name> in <normalised-column-list> must also be present as a <key-column-name> in <key-column-list>, followed by the keyword POINT.

5. If a <key-column-name> in <key-column-list> is not also a <normalised-column-name> in <normalised-column-list> then it may not be followed by the keyword POINT in the <key-column-list>.
5. If a <key-column-name> in <key-column-list> is not followed by either POINT or INTERVAL then INTERVAL is assumed.
6. NOT NULL must be declared in <other> for all the column names which are referenced either in <normalised-column-list> and / or <key-column-list>.

3.8 Incorporation of the Reformat and Normalise Operations

Description

A <query-spec> has been extended by a <reformat-clause> and a <normalise-clause> which enable the incorporation of reformat and normalise operations of VT-RA [01P 93]. Its syntax thus becomes

SELECT	<select-list>		(1)
FROM	<table-ref-list>		(2)
[WHERE	<search-condition>	]	(3)
[GROUP BY	<column-spec-list>	]	(4)
[HAVING	<search-condition>	]	(5)
[REFORMAT AS	<reformat-item>	]	(6)
[NORMALISE ON	<column-spec-list>	]	(7)
[ORDER BY	<column-spec-list>	]	(8)

Lines (1)-(5) are executed in the standard SQL sequence. We present the execution steps briefly and explain the new clauses.

Line 2 (Defines the tables in which data is stored).

Line 3 (Selects rows that satisfy a condition): The VT-SQL interval relational operators and scalar functions may also be used.

Line 4 (Groups rows on the basis of equal values in columns.)

Line 5 (Selects groups that satisfy a condition): The VT-SQL interval relational operators, scalar and aggregate functions may also be used.

Line 1 (Selects columns): The VT-SQL scalar and aggregate functions may also be used.

Line 6 (Reformats a table with respect to a sequence of columns of a time type): The REFORMAT AS implements the reformat operation of VT-RA [01P 93]. In particular, it introduces a sequence of unfold/fold operations which have to be performed on the table retrieved by the execution of the clauses in lines 1-5. Its syntax is as follows.

```
<reformat-item>     ::=     FOLD      <column-spec-list>  [<reformat-item>]
                          | UNFOLD    [ALL] <column-spec-list> [<reformat-item>]

<column-spec> ::=       <column-name>
                      | <table-name>.<column-name>
                      | <correlation-name>.<column-name>
                      | <unsigned-integer>
```

Some examples are the following:

```
REFORMAT    AS  FOLD  Time1, Time2

REFORMAT    AS  FOLD  R.Time1, 2

REFORMAT    AS  FOLD  1, 2

REFORMAT    AS  UNFOLD   Time1, 2
                FOLD     Time3

REFORMAT    AS  UNFOLD   Time1, 2
                FOLD     Time3, 1
                UNFOLD   Time4
```

Standard SQL allows duplicate rows. To maintain therefore compatibility with SQL, two versions of UNFOLD have been defined, UNFOLD and UNFOLD ALL. In the first case the table which is derived after a sequence of *unfolds* does not contain duplicate rows. In the second case the table may contain duplicate rows.

Line 7 (Normalises a table on certain columns of a time type): The clause

```
NORMALISE ON <column-spec-list>
```

implements the normalise operation of VT-RA [01P 93], that is, it is semantically equivalent to

```
REFORMAT AS
UNFOLD   <column-spec-list>
FOLD     <column-spec-list>
```

The rules which apply to <column-spec-list> are the same with those in the REFORMAT AS clause.

Line 8 (Sorts rows on the basis of columns): It is also possible to sort on columns of a time-interval type. In particular, assume, for demonstration reasons, that the table derived from the execution of clauses 1-7 has scheme R(A, Time) and contains the rows

(b, $[d_p, d_p)$)
(a, $[d_i, d_j)$)
(a, $[d_p, d_q)$)

Assume also that "$[d_p, d_q)$ *prequals* $[d_i, d_j)$" is satisfied. If the order clause is

ORDER BY A, Time

then these rows will be sorted as

(a, $[d_p, d_q)$)
(a, $[d_i, d_j)$)
(b, $[d_p, d_q)$)

Examples

"*Give the projects in which John was involved on each of the dates in [d3, d7)*."

```
SELECT          Project, intervsect(Time, '[d3, d7)')
FROM            PROJECT
WHERE           Name = 'John'
AND             Time cp '[d3, d7)'
REFORMAT AS
      UNFOLD    2
ORDER BY        2, Project
```

The execution of the first four lines of this query results in the table

Project	
P1	[d3, d5)
P2	[d3, d7)

Next, the REFORMAT AS clause is executed, which transforms it to

Project	
P1	d3
P1	d4
P2	d3
P2	d4
P2	d5
P2	d6

The execution of the last clause finally sorts the above table to

Project	
P1	d3
P2	d3
P1	d4
P2	d4
P2	d5
P2	d6

"*Give all disjoint time-intervals in which John was involved in some project, sorted in descending order*."

```
SELECT          Time
FROM            PROJECT
WHERE           Name = 'John'
REFORMAT AS
        FOLD    1
ORDER BY        1 DESC
```

The execution of the first three lines of the query results in the table

Time
[d1, d5)
[d2, d12)
[d15, d30)

which is next transformed to

Time
[d1, d12)
[d15, d30)

and is finally sorted as

Time
[d15, d30)
[d1, d12)

"*Normalise SHIFT on Hour, Day.*"

```
SELECT        Name, Day, Hour
FROM          SHIFT
NORMALISE ON  Hour Day
```

SHIFT

Name	Day	Hour
John	[d1, d5)	[h1, h9)
John	[d5, d10)	[h1, h12)
John	[d10, d20)	[h9, h12)

Consider the table below, which is not normalised, and the query "*Give the shift of every employee for each distinct day*".

SHIFT

Name	Day	Hour
John	[d2, d5)	[h1, h5)
John	[d3, d6)	[h4, h8)
John	[d4, d6)	[h3, h7)

The query

```
SELECT          Name, Day, Hour
FROM            SHIFT
   REFORMAT AS
        UNFOLD  Day
          FOLD  Hour
```

yields

Name	Day	Hour

John	d2	[h1, h5)
John	d3	[h1, h8)
John	d4	[h1, h8)
John	d5	[h9, h8)

The same result can be obtained if the query is formulated as

```
SELECT          Name, Day, Hour
FROM            SHIFT
    REFORMAT AS
        UNFOLD  Day
    NORMALISE ON Hour
```

"*Give the time-intervals of all the days with the property that in each day the sales exceeded 50000.*"

```
SELECT          Date
FROM            SALES
GROUP BY        Date
HAVING          sum(Qty*Price) > 50000
    REFORMAT AS
        FOLD    Day
    ORDER BY    Date
```

Format

```
<query-exp>     ::=   {<query-spec | <union-exp> | <except-exp>}[<order-clause>]

<query-spec>    ::=   SELECT [ALL | DISTINCT] <select-list> <table-exp>

<table-exp>     ::=   <from-clause>
                      [<where-clause>      ]
                      [<group-clause>      ]
                      [<having-clause>     ]
                      [<reformat-clause>   ]
                      [<normalise-clause>  ]

<reformat-clause>   ::=   REFORMAT AS <reformat-item>

<reformat-item>     ::=   FOLD <column-spec-list> [<reformat-item>]
                        | UNFOLD [ALL] <column-spec-list> [<reformat-item>]
```

<normalise-clause> ::= NORMALISE ON <column-spec-list>

<order-clause> ::= ORDER BY <order-item-list>

<order-item> ::= <order-column> [ASC | DESC]

<order-column> ::= <column-spec>
| <unsigned-integer>

<column-spec> ::= <column-name>
| <table-name>.<column-name>
| <correlation-name>.<column-name>
| <unsigned-integer>

The <union-exp> and <except-exp> in the <query-exp> are explained in sub-sections 3.9 and 3.10.

General Rule

1. Every SQL <query-spec> is also valid in VT-SQL.
2. A <query-spec> may have only one <order-clause> which is the last clause which is executed.

Additional Rules

1. A <query-spec> is executed as in standard SQL and it may incorporate the features of VT-SQL.
2. The <reformat-clause> is applied to the table derived after the execution of the SELECT statement.
3. The <normalise-clause> is applied to the table derived after the execution of the <reformat-clause>.
4. A <column-spec> must be referenced in the <select-list>.
5. The domain of a <column-spec> must be of a time type.
6. Before a sequence of FOLD operations is executed, duplicate rows are eliminated.
7. After a sequence of UNFOLD operations, duplicate rows are eliminated unless UNFOLD ALL has been specified.
8. The SQL expression x BETWEEN y and z does not apply if x, y, z are time-intervals.

Convention

The <reformat-clause> and the <normalise-clause> may not include columns of a time type on which either null time-points or null or semi-null time-intervals have been recorded.

3.9 Extension of Union

Description

The syntax of UNION has been extended so as to implement both the *union* operation of standard SQL and also the punion operation of VT-RA [01P 93]. In particular, if the syntax is

<query-spec-1> UNION <query-spec-2>

then a standard SQL UNION operation takes place. If the syntax is

<query-spec-1> UNION <reformat-column-list> <query-spec-2>

where <reformat-column-list> is a list of columns and we assume that <query-spec-1> and <query-spec-2> yield tables R1 and R2 respectively, then the operation is semantically equivalent to the VT-RA operation

R1 punion[<reformat-column-list>] R2

In either case <query-spec-1> and <query-spec-2> must yield union-compatible tables.

Examples

Assume that another table I1 is union-compatible with INFLATION and contains the tuples (A, 9.0, [m1, m7)) (A, 10.0, [m1, m13)). The query "*give the inflation of country A for all the time-intervals in INFLATION and I1, sorted by Time*" is formulated as

```
SELECT    Country, Percentage, Time
FROM      INFLATION
WHERE Country = 'A'
UNION
SELECT    Country, Percentage, Time
FROM      I1
WHERE     Country = 'A'
ORDER BY Time
```

This is a standard SQL query.

"*Give the greatest disjoint time-intervals in which either John's salary was 10K or he was assigned to project P2*."

```
SELECT    Time
FROM      SALARY
WHERE     Name = 'John'
AND       Amount = 10K
UNION     Time
SELECT    Time
FROM      PROJECT
WHERE     Name = 'John'
AND       Project  =  'P2'
```

This implements 'punion[Time]' and results in the single tuple ([d2, d12)).

Format

The syntax of this operation is given in the next sub-section, with the syntax of EXCEPT.

3.10 Direct Support of Operations Except and Pexcept

Description

In standard SQL the set-difference operation is not supported directly but it can be expressed by a nested query. In VT-SQL is supported directly for two reasons, firstly to achieve symmetry with operation pexcept of VT-RA, which is supported directly, and, secondly, to achieve symmetry with the direct support of the UNION operation of SQL [Date 86]. Specifically, one operation, EXCEPT, is defined in VT-SQL, which implements both the except and pexcept operations of VT-RA. In particular,

<query-spec-1> EXCEPT <query-spec-2>

implements except [01P 93]. If the syntax is

<query-spec-1> EXCEPT <reformat-column-list> <query-spec-2>

where <reformat-column-list> is a list of columns of a time type and we assume that <query-spec-1> and <query-spec-2> yield tables R1 and R2 respectively, then the operation is semantically equivalent to the VT-RA operation [01P 93]

R1 pexcept[<reformat-column-list>] R2

In all cases the two query specifications of EXCEPT must yield union-compatible tables.

Examples

"*Give employees who were not paid at any time in [d3, d10)*."

```
SELECT      Name
FROM        SALARY
GROUP BY    Name
EXCEPT
SELECT      Name
FROM        SALARY
WHERE       Time cp '[d3,  d10)'
GROUP BY    Name
```

"*Give the employees whose salary exceeded 9K at all times in [d5, d10)*."

```
SELECT      Name
FROM        SALARY
WHERE       Time cp '[d5,  d10)'
GROUP BY    Name
EXCEPT
SELECT      Name
FROM        SALARY
WHERE       Time cp '[d5, d10)'
AND         Amount <= 9K
GROUP BY    Name
```

"*Give the employees whose salary exceeded 9k at all times they were in the shoe department.*"

```
SELECT      S.Name
FROM        SALARY S, ASSIGNMENT A
WHERE       S.Name  =  A.Name
AND         S.Time cp A.Time
AND         A.Dept =  'shoe'
GROUP BY    Name
EXCEPT
SELECT      S.Name
FROM        SALARY S, ASSIGNMENT A
WHERE       S.Name = A.Name
AND         S.Time cp A.Time
AND         A.Dept = 'shoe'
```

```
AND         S.Amount <= 30K
GROUP BY    Name
```

The above examples represent demonstrations of operation *except*. Examples to demonstrate *pexcept* are the following.

"*Give the employees and the time-intervals in which they were in the shoe department, excluding the time during which John was also working in it.*"

```
SELECT      Name, Time
FROM        ASSIGNMENT A
WHERE       A.Name <> 'John'
AND         A.Dept = 'shoe'
EXCEPT      Time
SELECT      A1.Name, intervsect(A1.Time, A2.Time)
FROM        ASSIGNMENT A1, ASSIGNMENT A2
WHERE       A1.Name <> 'John'
AND         A1.Dept = 'shoe'
AND         A1.Time cp A2.Time
AND         A2.Name = 'John'
AND         A2.Dept = 'shoe'
```

"*Give John's shift for each of the dates d5-d9, excluding the data in the next table.*"

SHIFT1

Day	Hour
[d6, d8)	h5
[d6, d8)	h6
[d6, d8)	h7

```
SELECT          intervsect(Day, '[d5, d10)'), Hour
FROM            SHIFT
WHERE           Name = 'John'
AND             Day cp '[d5, d10)'
   REFORMAT AS
        UNFOLD  1
EXCEPT          1, 2
SELECT          Day, tointerv(Hour)
FROM            SHIFT1
   REFORMAT AS
        UNFOLD  1
```

Assume that the enterprise is operational for all the time at which at least one employee is paid. Now consider the query "*give all the time-intervals at which the enterprise was not operational*".

```
SELECT   interv(min(start(Time), max(stop(Time))
FROM     SALARY
EXCEPT   1
SELECT   Time
FROM     SALARY
```

Format

```
<query-exp> ::=     {<query-spec> | <union-exp> | <except-exp>}
                    [<order-clause>]

<query-spec> ::=    SELECT [ALL | DISTINCT] <select-list> <table-exp>

<union-exp> ::=     <query-spec>
                    UNION [{ ALL | <reformat-column-list> }]
                    <query-spec>

<except-exp> ::=      <query-spec>
                      EXCEPT [<reformat-column-list>]
                      <query-spec>

<table-exp> ::=     <from-clause>
                   [<where-clause>          ]
                   [<group-clause>          ]
                   [<having-clause>         ]
```

[<reformat-clause>]
[<normalise-clause>]

<reformat-column> ::= <column-spec>
| <unsigned-integer>

<column-spec> ::= <column-name>
| <table-name>.<column-name>
| <correlation-name>.<column-name>

General Rules

1. Every SQL query expression which involves UNION is also valid in VT-SQL.
2. A <query-spec> may have only one <order-clause> which is the last clause which is executed.

Additional Rules

1. The lines in SELECT and <table-exp> are executed as explained earlier.
2. The two <query-spec> in <union-exp> and <except-exp> must yield union-compatible tables.
3. Before EXCEPT is executed, duplicate data in the <query-exp> which precedes EXCEPT has to be eliminated.

Convention

The <reformat-column-list> in both UNION and EXCEPT may not include columns of a time type on which either null time-points or null or semi-null time-intervals have been recorded.

3.11 Extension to the Data Manipulation Statements

The fact that the SQL2 CREATE TABLE command has been extended in VT-SQL by the incorporation of the NORMALISED clause, now simplifies the syntax of the data manipulation statements. Each of them is described separately next. The examples which are provided should be seen in conjunction with section 3.7, where it has been shown how certain tables have been declared to the DBMS.

3.11.1 Data Insertion

Description

The syntax of the SQL2 INSERT statement remains exactly the same in VT-SQL. However, according to the explanations already provided in the CREATE TABLE statement above, the way data is inserted, depends completely on how a table has been declared to the DBMS. This is demonstrated by the following examples.

Examples

"*Insert into INFLATION the row* (A, 10.0, [m7, m13))."

```
INSERT
INTO      INFLATION(Country, Percentage, Time)
VALUES    ('A', 10.0, '[m7, m13)')
```

We notice that the definition of table INFLATION does not contain a NORMALISED clause. This implies that INSERT functions in exactly the same way as in SQL2 that is, it results in a table which is semantically equivalent to

```
INFLATION = INFLATION union S
```

thus yielding

INFLATION

Country	Percentage	Time
A	8.0	[m1, m4)
A	10.0	[m4, m7)
A	10.0	[m7, m13)
A	10.0	[m1, m13)

It should be noted that if the key of a table has been declared, then in SQL2 an insertion fails in either of the following cases: (i) The key values of one of the rows to be inserted in R matches the respective values of one of the rows already recorded in R. (ii) Two of the rows to be inserted in R have the same value for the key columns. In either of these cases, the insertion fails completely. To provide examples, assume that the key of INFLATION has been declared and assume that we attempt to insert either of the following set of rows:

(a) (A, 8.0, [m1, m4))
(A, 9.0, [m1, m7))

(Case (i) above, the first of them has already been recorded.)

(b) (A, 9.0, [m1, m4))
(A, 9.0, [m1, m7))

(Case (i) above, key violation.)

(c) (A, 9.0, [m1, m7))
(A, 9.0, [m1, m7))
(A, 12.0, [m7, m13))

(Case (ii) above, identical rows.)

(d) (A, 9.0, [m1, m7))
(A, 12.0, [m7, m13))
(A, 13.0, [m7, m13))

(Case (ii) above, two rows violate the key constraints.)

Then for any of cases (a)-(d), nothing is inserted in INFLATION.

For compatibility reasons, this functioning of the SQL2 INSERT statement is not only preserved in VT-SQL but also extended appropriately. In particular, assume that we attempt to insert into SALARY any of the following set of rows:

(a) (John, 10K, [d2, d6))
(Mary, 10K, [d2, d6))

(Case (i), the first of them has already been recorded.)

(b) (John, 10K, [d5, d8))
(Mary, 10K, [d2, d6))

(Case (i), the data for John's salary on date d5 has already been recorded.)

(c) (John, 10K, [d6, d8))
(John, 10K, [d6, d9))
(Mary, 10K, [d2, d6))

(Case (ii), John's salary for dates d6 and d7 is recorded in two rows.)

(d) (John, 10K, [d6, d8))
(John, 11K, [d6, d9))
(Mary, 10K, [d2, d6))

(Case (ii), John's salary for dates d6 and d7 violates the key.)

Then in all (a)-(d) cases above, nothing will be inserted in SALARY. Some examples to demonstrate the functioning of VT-SQL INSERT, are the following:

"*Insert into SALARY the data (John, 10K, [d3, d10).*"

We initially assume that the key of SALARY has not been declared. The command is issued as

```
INSERT
INTO     SALARY
VALUES   ('John' 10K, '[d3, d10)')
```

We now recall that the declaration of SALARY includes "NORMALISED (Time)". As a consequence, at insertion the new data is normalised with the data already recorded in SALARY, thus implementing

SALARY = SALARY punion[Time] S

where S is a constant table consisting of the above tuple, and yields

SALARY

Name	Amount	Time
John	10K	[d2, d12)
John	12K	[d15, d18)

Alex	14K	[d9, d12)

It should be noted that John's salary for dates d3, d4, d5 and d9, which was contained in S, had already been recorded into SALARY. Given however that the key of SALARY had not been declared to the DBMS, the insertion command was executed without any problem. Yet, if we assume that the key of SALARY has been declared then, as reported above, the command will fail.

Is should be noted that the VALUES(<values-list>) in the above example may be replaced by a <query-spec>.

Format

<insert-stat> ::= INSERT
INTO <table-name> [(<column-ref-list>)]
<source-values>

<source-values> ::= VALUES (<values-list>) | <query-spec>

<values> ::= <literal> | NULL

General Rules

1. Every valid INSERT statement of standard SQL, is also valid in VT-SQL

Additional Rules

1. The data to be inserted in <table-name> has to be compatible with the data in this table.
2. If the table definition contains a NORMALISED <normalised-column-list> then a normalisation always takes place in data insertion.
3. If the key of a table R has been declared to the DBMS and either (i) the key values of one of the rows to be inserted in R matches the respective values of one of the rows already recorded in R or (ii) two of the rows to be inserted in R have the same value for the key columns then the insertion command fails completely.

Conventions

1. The value of every piece of data which is to be inserted in a table may not contain null or semi-null time-intervals on components with respect to which a normalization takes place.

2. The value of every piece of data which is to be inserted in a table may not contain null or semi-null time-intervals on components which participate to the key.

3.11.2 Data deletion

Description

The syntax of the DELETE statement of standard SQL is

DELETE FROM <table-name>
[<where-clause>]

but this does not enable a satisfactory formulation of commands for the deletion of data from tables normalised with respect to some of their columns. We have thus extended this syntax as

DELETE FROM <table-name>
[PORTION <normalised-column-value-list>]
[<where-clause>]

where
<normalised-column-value> :: = <normalised-column-name> = <value-exp>
To demonstrate its functionality by an example, let R(A, B) be a table normalised with respect to B and assume that

DELETE FROM R
PORTION B = '[d5, d15)'
WHERE A = 'a'

has been issued. Let also (a, [d1, d20)) be one of the tuples of R which is to be deleted. We then notice that this tuple can be *split* into the tuples

(a, [d1, d5))
(a, [d5, d15))
(a, [d15, d20))

the second of which contains the interval in the list after the keyword PORTION of the above deletion statement. After the execution of the statement, R will contain the tuples

(a, [d1, d5))
(a, [d15, d20))

in place of tuple (a, [d1, d20)). Therefore, the extended statement is applied to a normalised table R and deletes *from each tuple of R which satisfies the deletion criteria, that portion of data which is explicitly referenced in the list after the keyword PORTION*.

For a formal definition, we consider below two distinct cases:

Case (i): It has *not* been declared in the CREATE TABLE <table-name> statement that <table-name> is normalised.

If R is such a non-normalised table then the syntax of the delete statement is

DELETE FROM R
[<where-clause>]

exactly as in standard SQL. If we assume that another table S consists of the rows of R which satisfy the deletion criteria, the result obtained is semantically to

R = R except S

Case (ii): It has been declared in the CREATE TABLE <table-name> statement that <table-name> is normalised.

Assume that the scheme of such a table is $R(A_1, A_2, ..., A_p, B_1, B_2, ..., B_q)$ and that it has been declared in the CREATE TABLE statement that R is normalised on all the B_i columns. If $\{W_1, W_2, ..., W_r\} \subseteq \{B_1, B_2, ..., B_q\}$, we define that the result obtained by

DELETE FROM R
[PORTION $W_1 = w_1, W_2 = w_2, ..., W_r = w_r$]
[<where-clause>]

is semantically equivalent to the following sequence of steps:

Step 1: Let S be a table consisting of the rows of R which satisfy

(<where-clause>) and (W_1 *cp* w_1 and W_2 *cp* w_2 and ... and W_r *cp* w_r)

(If the <where-clause> or "PORTION $W_1 = w_1, W_2 = w_2, ..., W_r = w_r$"or both of them are missing, this search statement is adjusted appropriately.)

Step 2: Replace each $S.W_i$ value of S by the respective *intervsect* $(S.W_i, w_i)$.

Step 3: R = R pexcept$[C_1, C_2, ..., C_q]$ S

We notice that after the deletion, R remains normalised with respect to $[C_1, C_2, ..., C_q]$.

Examples

"*Delete from INFLATION the inflation of country A for* [m1, m13)."

```
DELETE FROM     INFLATION
WHERE           Country = 'A'
```

```
AND                 Time    =  '[m1,  m13)'
```

Since it has not been declared that INFLATION is normalised, case (i) above applies, and a standard SQL deletion takes place.

"*Delete from SALARY John's data for the time-interval* [d2, d6)."

```
DELETE FROM     SALARY
PORTION         Time    =  '[d2,  d6)'
WHERE           Name    =  'John'
```

Here it is only a coincidence that [d2, d6) is an interval explicitly recorded in SALARY. The result is that the first row of SALARY is eliminated.

"*Delete the data for John's salary during the interval [d5, d10)*."

The query represents the general case of a deletion from a table which has been normalised with respect to certain columns. We now want to eliminate John's data during an arbitrary time-interval, not explicitly recorded on column Time of some row of SALARY..

```
DELETE FROM     SALARY
PORTION         Time    =  '[d5,  d10)'
WHERE           Name    =  'John'
```

SALARY

Name	Amount	Time
John	10K	[d2, d5)
John	10K	[d10, d12)
John	12K	[d15, d18)
Alex	14K	[d9, d12)

"*Delete from SALARY all the data for John*."

```
DELETE
FROM    SALARY
WHERE   Name = 'John'
```

We notice that PORTION <normalised-column-value-list> is not necessary. The command eliminates all the rows for John.

"*Purge all the data from SALARY until time d10*."

```
DELETE FROM    SALARY
PORTION        Time = '[mindate, d11)'
```

SALARY

Name	Amount	Time
John	10K	[d11, d12)
John	12K	[d15, d18)
Alex	14K	[d11, d12)

"Delete all the data from SALARY"

```
DELETE FROM    SALARY
```

"*Delete from SHIFT John's data for each of the hours in [h5, h10) in each of the dates in* [d6, d8)."

```
DELETE FROM    SHIFT
PORTION        Hour = '[h5, h10)', Day = '[d6, d8)'
WHERE          Name = 'John'
```

SHIFT

Name	Day	Hour
John	[d1, d10)	[h1, h5)
John	[d1, d6)	[h5, h9)
John	[d8, d10)	[h5, h9)
John	[d5, d6)	[h9, h10)
John	[d8, d20)	[h9, h10)
John	[d5, d20)	[h10, h12)

"*Delete from SHIFT John's data for each of the dates in* [d6, d8)."

```
DELETE FROM    SHIFT
PORTION        Day = '[d6, d8)'
```

WHERE Name = 'John'

The examples shows that it is not necessary for all the columns on which a table has been normalised need appear after the keyword PORTION.

SHIFT

Name	Day	Hour
John	[d1, d6)	[h1, h9)
John	[d8, d10)	[h1, h9)
John	[d5, d6)	[h9, h12)
John	[d8, d20)	[h9, h12)

Format

<delete-stat> : := DELETE FROM <table-name>
[PORTION <normalised-column-value-list>]
[<where-clause>]

<normalised-column-value> ::= <normalised-column-name> = <value-exp>

General Rules

1. Every valid DELETE statement of standard SQL, is also valid in VT-SQL.

Additional Rules

1. PORTION <normalised-column-value-list> may appear only if it has been declared in the CREATE TABLE statement that <table-name> is normalised.
2. Every <normlalised-column-name> in <normalised-column-value-list> must be the name of a column on which it has been declared in the CREATE TABLE statement that <table-name> is normalised.
3. A <normalised-column-name> may not appear more than once in <normalised-column-value-list>.
4. Each <value-exp> in <normalised-column-value-list> must evaluate to an interval compatible with the domain of the proceeding <normalised-column-name>.

Conventions

1. A <value-exp> may not be a null or semi-null interval.

3.11.3 Data Update

Description

The syntax of the UPDATE statement of standard SQL is

UPDATE <table-name>
SET <column-assignment-list>

[<where-clause>]

but this does not enable a satisfactory formulation of commands for the update of tables which have been normalised with respect to some of their columns. We have thus extended it in a consistent way, as

UPDATE <table-name>
[PORTION <normalised-column-value-list>]
SET <column-assignment-list>
[<where-clause>]

where
<normalised-column-value> :: = <normalised-column-name> = <value-exp>

To demonstrate its functionality by an example, let R(A, B) be a table normalised with respect to B and assume that

UPDATE <table-name>
PORTION B = '[d5, d15)'
SET A = 'a2'
[<where-clause>]

has been issued. Let also (a1, [d1, d20)) be one of the tuples of R which is to be updated. We then notice that this tuple can be *split* into the tuples

(a1, [d1, d5))
(a1, [d5, d15))
(a1, [d15, d20))

the second of which contains the interval which is next to the keyword PORTION of the above deletion statement. After the execution of the above statement, R contains the tuples

(a1, [d1, d5))
(a2, [d5, d15))

(a1, [d15, d20))

in place of the tuple (a1, [d1, d20)). Therefore, for each row of R which satisfies the update criteria, the extended statement updates *that portion which is explicitly referenced after the keyword PORTION*..

For a formal definition, we consider below two distinct cases:

Case (i): It has *not* been declared in the CREATE TABLE <table-name> statement that <table-name> is normalised.

Let R be a non-normalised table. Then the syntax of the statement which updates it, is

UPDATE R
SET <column-assignment-list>
[<where-clause>]

exactly as in standard SQL, and it is semantically equivalent to the following sequence of steps:

Step 1: Let S be the rows of R which satisfy the update criteria.

Step 2: R = R except S

Step 3: For each assignment *A=a* in <column-assignment-list> *replace* by *a* the value of each row of S for column A.

Step 4: R = R union S

Case (ii): It has been declared in the CREATE TABLE <table-name> statement that <table-name> is normalised.

Assume that the scheme of such a table is $R(A_1, A_2, ..., A_p, B_1, B_2, ..., B_q)$ and that it has been declared in the CREATE TABLE statement that it is normalised on all the B_i columns. If $\{W_1, W_2, ..., W_r\} \subseteq \{B_1, B_2, ..., B_q\}$, we define that the result obtained by

DELETE FROM R
[PORTION $W_1 = w_1,\ W_2 = w_2, ...,\ W_r = w_r$]
[<where-clause>]

is semantically equivalent to the following sequence of steps:

Step 1: Let S be a table, union-compatible to R, consisting of the rows of R which satisfy

(<where-clause>) and (W_1 *cp* w_1 and W_2 *cp* w_2 and ... and W_r *cp* w_r)

(If the <where-clause> or PORTION $W_1 = w_1,\ W_2 = w_2, ...,\ W_r = w_r$ or both of them are missing, this search statement is adjusted appropriately.)

Step 2: Replace each $S.W_i$ value of S by the respective *intervsect*$(S.W_i, w_i)$.

Step 3: R = R pexcept$[B_1, B_2, ..., B_q]$ S

Step 4: For each assignment *A=a* in <column-assignment-list> *replace* by *a* the value of each row of S for column A.

Step 5: R = R punion$[B_1, B_2, ..., B_q]$ S

We notice that after the execution of UPDATE, R remains normalised with respect to $[C_1, C_2, ..., C_q]$.

Examples

"*Replace the time-interval [m1, m4) of country A by the correct one,* [m1, m7)".
Since it has not been declared that INFLATION is normalised with respect to any of its columns, the command is formulated exactly as in standard SQL:

```
UPDATE    INFLATION
SET       Time    =  '[m1, m7)'
WHERE     Country =  'A'
AND       Time    =  '[m1, m4)'
```

"*Update SALARY that it was not John's salary 10K but Mary's.*"

```
UPDATE    SALARY
SET       Name    =  'Mary'
WHERE     Name    =  'John'
AND       Amount  =  10K
```

The example shows that even if a table is normalised, PORTION <normalised-column-value-list> is not necessary.

SALARY

Name	Amount	Time
Mary	10K	[d2, d6)
Mary	10K	[d9, d12)
John	12K	[d15, d18)
Alex	14K	[d9, d12)

"*It has been recorded by mistake that John's salary was 10K for each of the days in* [d10, d12). *Update the data to 11K for this time-interval*".

```
UPDATE      SALARY
PORTION     Time     =   '[d10, d12)'
SET         Amount   =   11K
WHERE       Name     =   'John'
AND         Amount   =   10K
AND         Time     cp  '[d10, d12)'
```

We notice that [d10, d12) is not an interval explicitly recorded in column Time of some row of SALARY.

SALARY

Name	Amount	Time
John	10K	[d2, d6)
John	10K	[d9, d10)
John	11K	[d10, d12)
John	12K	[d15, d18)
Alex	14K	[d9, d12)

"*It has been recorded by mistake a salary for John, for each of the days in* [d11, d12). *Update the data to the correct one, that the salary was 12K, and the time-interval was* [d18, d25)."

```
UPDATE      SALARY
PORTION     Time     =   '[d11, d12)'
```

```
SET       Amount  =  12K, Time ='[d18, d25)'
WHERE     Name    =  'John'
AND       Time    cp '[d11, d12)'
```

SALARY

Name	Amount	Time
John	10K	[d2, d6)
John	10K	[d9, d11)
John	12K	[d15, d25)
Alex	14K	[d9, d12)

"*The data concerning John's salary for each of the days in* [d9, d18) *is not correct. The correct is that each of the days have to be restricted to those in* [d10, d16)."

```
UPDATE    SALARY
PORTION   Time    =  '[d9, d18)'
SET       Time    =  '[d10, d16)'
WHERE     Name    =  'John'
AND       Time    cp '[d9, d18)'
```

SALARY

Name	Amount	Time
John	10K	[d2, d6)
John	10K	[d10, d12)
John	12K	[d15, d16)
Alex	14K	[d9, d12)

"*It has been recorded by mistake John's shift for each of the dates in* [d10, d20)*. Update it to the correct, that this is Alex's shift.*"

```
UPDATE    SHIFT
PORTION   Day     =  '[d10, d20)'
SET       Name    =  'Alex'
WHERE     Name    =  'John'
AND       Time    cp '[d10, d20)'
```

SHIFT

Name	Day	Hour
John	[d1, d10)	[h1, h9)
John	[d5, d10)	[h9, h12)
Alex	[d10, d20)	[h9, h12)

We now recall that in SQL2 an update operation of a table R fails completely in any of the following cases: (i) The value for the key columns of one of the updated rows matches the respective values of one of the rows already recorded in R. (ii) Two of the updated rows have the same value for the key columns. For compatibility reasons, this rule is also maintained in VT-SQL, as is shown by the next example:

"*Update SALARY to record that 10K was Alex's salary and not John's.*"

```
UPDATE    SALARY
SET       Name   = 'Alex'
WHERE     Name   = 'John'
AND       Amount = 10K
```

If no primary key had been declared in the CREATE TABLE SALARY statement, the result would be

SALARY

Name	Amount	Time
John	12K	[d15, d18)
Alex	10K	[d2, d6)
Alex	10K	[d9, d12)
Alex	14K	[d9, d12)

If hovever the primary key has been declared then the update statement will be completely rejected because, for example, this would result in that Alex's salary for date d9 would be recorded in SALARY twice.

Format

```
<update-stat> ::=  UPDATE <table-name>
                   PORTION <normalised-column-value-list>]
                   SET <column-assignment-list>
```

<normalised-column-value> ::= <normalised-column> = <value-exp>

<column-assignment> ::= <column-ref> = { <scalar-exp> | NULL }

General Rules

1. Every valid UPDATE statement of standard SQL, is also valid in VT-SQL.

Additional Rules

1. The data which is to replace that in <table-name> has to be compatible with the data in <table-name>.
1. PORTION <normalised-column-value-list> may appear only if it has been declared in the CREATE TABLE statement that <table-name> is normalised.
2. Every <normlalised-column-name> in <normalised-column-value-list> must be the name of a column on which it has been declared in the CREATE TABLE statement that <table-name> is normalised.
3. A <normalised-column-name> may not appear more than once in <normalised-column-value-list>.
4.. Each <value-exp> in <normalised-column-value-list> must evaluate to an interval compatible with the domain of the proceeding <normalised-column-name>.
5. If the primary key of <table-name> has been declared in the CREATE TEBLE <table-name> statement then the UPDATE statement fails completely in each of the following cases: (i) A piece of data for the key columns of one of the updated rows has already recorded in <table-name>. (ii) Two of the updated rows have the same piece of data for the key columns.

Conventions

1. A <value-exp> may not be a null or semi-null interval.

Remark

Optimization techniques will be incorporated at the implementation of the VT-SQL statements INSERT, UPDATE and DELETE.

3.12 *Other VT-SQL Statements*

From the remainder SQL statements we notice the following.

Create Index Statement

It remains exactly the same as in standard SQL.

Create View

In a complete implementation, provision has to be made for the support of views which incorporate the features of VT-SQL. This is however a feature which requires further investigation related mainly to whether a view is updatable or not.

Grant Statement

It remains the same.

4. *COMMENTS ON VT-SQL*

VT-SQL is a consistent extension to standard SQL in that it maintains its syntax and semantics. At the same time it supports all the VT-RA operations. Some comments on the specification of VT-SQL are the following.

(i) It would be desirable to allow nested queries in a <from-clause> of VT-SQL. If this were allowed, certain VT-SQL queries would not have to be broken into more than one query. However, for symmetry reasons this nesting should also be allowed to pure SQL queries. However, SQL is not orthogonal [Date 86] and the aim of ORES is far from defining an orthogonal extension to SQL. A relevant example in SQL is the query

```
SELECT max(T.City)
FROM   (SELECT S.City FROM S
        UNION
        SELECT P.City FROM P)
        AS  T(City)
```

which is not valid in fact. To obtain the result targeted by it, one has to formulate two distinct SQL queries.

(ii) It could have been argued that a <normalise-clause> is not necessary in a query specification which precedes or follows a UNION or an EXCEPT. We have allowed it for the reasons described next.

Symmetry: If SHIFT1 is the table

SHIFT1

Name	Day	Hour
john	d6	[h5, h12)
john	d7	[h6, h12)
john	d8	[h7, h12)

then it does make sense to issue the command

```
INSERT          Hour
INTO            SHIFT1
SELECT          Name, Day, Hour
FROM            SHIFT
REFORMAT AS
        UNFOLD  Day
```

(Indeed, if this command were not allowed, one would firstly have to unfold SHIFT on Day and obtain the result in a table R and then issue the INSERT statement to insert the contents of R into SHIFT1.) Since there is a symmetry between UNION and INSERT, it is principally necessary for a <reformat-clause> to be allowed to any of the query specifications which precede or follow UNION or EXCEPT. Finally, since NORMALISE is only a special case of REFORMAT, it is also reasonable, again for symmetry reasons, for NORMALISE to be applied to any of the two query specifications which surround a UNION or EXCEPT.

Easiness in query formulation: Queries like the following ones can be formulated in one step.

```
SELECT          A, B, C
FROM            S1
UNION
SELECT          A, B, C
FROM            S2
REFORMAT AS
        UNFOLD  C
```

(REFORMAT is necessary for the two tables to become union-compatible.)

```
SELECT          A, B, C, D, E
FROM            S1
NORMALISE ON    A, B
UNION           D, E
SELECT          A, B, C, D, E
FROM            S2
REFORMAT AS
        UNFOLD  C
```

(Before *punion* is applied, they become union-compatible.)

Combination of the above: In contrast with the <order-clause>, the <reformat-clause> and <normalise-clause> may be in a nested query and this enables the user formulate queries in the way he finds it most convenient.

For the above reasons, it has been determined that discarding a redundant normalisation should rather be a task of the DBMS optimiser.

5. *VT-SQL AND THE ORES PROJECT*

Although it would be desirable to reach a complete implementation of VT-SQL, it is anticipated that this is really difficult for a number of reasons, the most serious of which are the short life span of the project, in conjunction with the substantial programming effort which is required. In particular, the implementation issues which can hardly be attacked are the following.

(i) The support of a <normalise-clause> or a <reformat-clause> in a sub-query, for example

```
SELECT  A, B
FROM    S1
WHERE   '[d5,  d10)' =  (SELECT          A
                         FROM            S2
                         NORMALISE ON    A)
```

It is a desirable property because it may simplify the formulation of certain queries. In this case however we shall deprive the optimisation capabilities of INGRES. It should be noted that if the support of sub-queries of this type is not supported, no major problem will arise. In particular, an initial investigation, which we have undertaken, has shown that queries which involve a <normalise-clause> or <reformat-clause> in a sub-query, can equivalently be expressed in ways by which such clauses can be eliminated.

(ii) The use of all the interval relational operators before a sub-query, either in a <where-clause> or in a <having-clause> for example,

```
SELECT  A, B
FROM    S1
WHERE   '[d5, d10)' adjacent   (SELECT   A
                                FROM     S2
                                <where-clause>)
```

Our investigation has shown that the Object Manager of INGRES does not allow the use of user-defined relational operators before sub-queries. One alternative solution which we have investigated, is for the user to formulate a query in a way like the above and, before execution, the query to be transformed to the equivalent one

```
SELECT  A, B
FROM    S1
WHERE   stop('[d5, d10)')  =  (SELECT start(A)
                              FROM   S2
                              <where-clause>)
OR      start('[d5, d10)')  =  (SELECT stop(A)
                              FROM   S2
                              <where-clause>)
```

This is possible for all the relational operators of VT-SQL. This solution however will increase the execution time. In particular, our investigation has shown that, depending on the operator, a user sub-query has to be transformed up to three SQL sub-queries with the same <where-clause>. It is therefore obvious that the execution time will increase exponentially with respect to the nesting depth of a query.

(iii) The support of the two VT-SQL aggregate functions. Since aggregate functions may appear in many places in a query, this would require the development of a VT-DBMS almost from scratch. It should be noted however, that if aggregate functions are not supported, the user will again be able to obtain the results returned by them, by formulating the queries in other equivalent ways which do not involve aggregate functions.

In addition to the above, the following should be taken into consideration.

(i) VT-SQL is more than the minimal extension to SQL, which the ORES Technical Annex requires.

(ii) The partial implementation of VT-SQL, as specified next, completely satisfies the requirements of the test bed application [CPH 93].

(iii) The project's life span is too short.

The software which will be developed will include the following.

(i) The support of a DATEINTERVAL type.

(ii) The support of the VT-SQL relational operators and scalar functions in all other places except those specified above.

(iii) The support of the <reformat-clause> and <normalise-clause> at a non-nested query, applied to one time-point or time-interval. In addition to the ORES Technical Annex, we are planning to include some optimisation here.

(iv) The support of EXCEPT and the extended version of UNION, with some optimisation. Queries will be of the type

<query-spec> [UNION | EXCEPT] <query-spec>.

(v) The support of the extended versions of INSERT, DELETE, and UPDATE, with some optimisation.

Effort will also be made to support the following:

(i) The combination of the DATEINTERVAL type with the INGRES DATE type.

(ii) The definition of other time-interval types.

(iii) The enforcement of the semantics of VT-SQL.

(iv) The application of the <reformat-clause> and <normalise-clause> at a non-nested query, applied to more than one time-point or time-interval. (Note that this really make sense only in the case that a table contains at least two columns of distinct time types.).

The syntax of VT-SQL will probably be slightly different in the implementation, if we are faced with constraints imposed by INGRES.

6. CONCLUSIONS

In this report we gave the specification of a valid time extension to standard SQL. It is a consistent extension of SQL, in that it preserves its syntax and semantics. Its definition has been based on the Valid Time Relational Algebra (VT-RA) [01P 93], whose operations are fully supported.

APPENDIX A

FORMAL DEFINITIONS

PREDEFINED CONSTANTS

mindate: It equals the least date supported by a VT-DBMS.

maxdate: It equals the greatest date supported by a VT-DBMS.

AGGREGATE FUNCTIONS

countap : (*Count all time-points*) returns the number of time-points that are contained in a selected column or value expression. Duplicate points are also counted.

countdp : (*Count distinct time-points*) returns the number of *distinct* time-points that are contained in a selected column or value expression.

PREDICATES

dy	*before*	dx :	*stop*(dy)	<	*start*(dx)		
dy	*meets*	dx :	*stop*(dy)	=	*start*(dx)		
dy	*loverlaps*	dx :	*start*(dy)	<	*start*(dx)	**and**
			stop(dy)	>	*start*(dx)	**and**
			stop(dy)	<	*stop*(dx)	
dy	*lcovers*	dx :	*start*(dy)	<	*start*(dx)	**and**
			stop(dy)	=	*stop*(dx)	
dy	*covers*	dx :	*start*(dy)	<	*start*(dx)	**and**
			stop(dy)	>	*stop*(dx)	
dy	*rcovered*	dx :	*start*(dy)	=	*start*(dx)	**and**
			stop(dy)	<	*stop*(dx)	
dy	=	dx :	*start*(dy)	=	*start*(dx)	**and**
			stop(dy)	=	*stop*(dx)	

dy *rcovers* dx : *start*(dy) = *start*(dx) **and**
stop(dy) > *stop*(dx)

dy *covered* dx : *start*(dy) > *start*(dx) **and**
stop(dy) < *stop*(dx)

dy *lcovered* dx : *start*(dy) > *start*(dx) **and**
stop(dy) = *stop*(dx)

dy *roverlaps* dx : *start*(dy) > *start*(dx) **and**
start(dy) < *stop*(dx) **and**
stop(dy) > *stop*(dx)

dy *met* dx : *start*(dy) = *stop*(dx)

dy *after* dx : *start*(dy) > *stop*(dx)

dy *psubinterv* dx : *start*(dy) >= *start*(dx) **and**
stop(dy) <= *stop*(dx) **and**
not (
start(dy) = *start*(dx) **and**
stop(dy) = *stop*(dx)
)

dy *subinterv* dx : *start*(dy) >= *start*(dx) **and**
stop(dy) <= *stop*(dx)

dy *psupinterv* dx : *start*(dy) <= *start*(dx) **and**
stop(dy) >= *stop*(dx) **and**
not (
start(dy) = *start*(dx) **and**
stop(dy) = *stop*(dx)
)

dy *supinterv* dx : *start*(dy) <= *start*(dx) **and**
stop(dy) >= *stop*(dx)

```
dy  overlaps        dx : not (
                         not (
                             start(dy) < start(dx) and
                             stop(dy) > start(dx) and
                             stop(dy) < stop(dx)
                             )
                         and
                         not (
                             start(dy) > start(dx) and
                             stop(dy) > stop(dx)
                             )
                         )

dy  merges          dx : stop(dy) >= start(dx)      and
                         start(dy) <= stop(dx)

dy  cp              dx : stop(dy) > start(dx)       and
                         start(dy) < stop(dx)

dy  precedes        dx : not (
                         not (
                             start(dy) < start(dx) and
                             stop(dy) <= stop(dx)
                             )
                         and
                         not (
                             start(dy) = start(dx) and
                             stop(dy) < stop(dx)
                             )
                         )

dy  follows         dx : not (
                         not (
                             start(dy) > start(dx) and
                             stop(dy) >= stop(dx)
                             )
                         and
                         not (
                             start(dy) = start(dx) and
```

stop(dy) > *stop*(dx)
)
)

dy *prequals* dx : **not** (
not (
start(dy) < *start*(dx) **and**
stop(dy) <= *stop*(dx)
)
and
not (
start(dy) = *start*(dx) **and**
stop(dy) <= *stop*(dx)
)
)

dy *folequals* dx : **not** (
not (
start(dy) > *start*(dx) **and**
stop(dy) >= *stop*(dx)
)
and
not (
start(dy) = *start*(dx) **and**
stop(dy) >= *stop*(dx)
)
)

dy *adjacent* dx : **not** (
stop(dy) <> *start*(dx) **and**
start(dy) <> *stop*(dx)
)

FUNCTIONS

The following notations are used in the definition of the functions incorporated in VT-SQL.

I: The set of integeres

R: The set of reals

$D = \{d_1, d_2, ..., d_n\}$: A set of consecutive dates.

$I(D) = \{(d_i, d_j) \mid d_i, d_j \in D, d_i < d_j\}$: A set of intervals over D.

trunc: integer part of a real number.

div: quotient of an integer division.

now:	Returns the current date	
start:	$I(D) \rightarrow D$:	$start([d_i, d_j)) = d_i$
stop:	$I(D) \rightarrow D$:	$stop([d_i, d_j)) = d_j$
topoint:	$I(D) \rightarrow D$:	$topoint([d_i, d_{i+1})) = di$
tointerv:	$D \rightarrow I(D)$:	$tointerv(d_i) = [d_i, d_{i+1})$
interv:	$D \times D \rightarrow I(D)$:	$interv(d_i, d_j) = [d_i, d_j), i < j$
intervsect:	$I(D) \times I(D) \rightarrow I(D)$:	
	$intervsect([d_i, d_j), [d_p, d_q)) = [max(\{d_i, d_p\}), min(\{d_j, d_q\}))$, $[d_i, d_j)\ cp\ [d_p, d_q) = True$	
succ:	$D \times I \rightarrow D$:	$succ(d_i, k) = d_{i+k}, 1 \le i + k \le n$
dur:	$I(D) \rightarrow I$:	$dur([d_i, d_j)) = j - i$
span:	$D \times D \rightarrow I$:	$span(d_i, d_j) = i - j$
dist:	$D \times D \rightarrow I$:	$dist(d_i, d_j) = \mid i - j \mid$
middle:	$I(D) \rightarrow D$:	$middle([d_i, d_j)) = d_{i+trunc((j-i)/2)}$
merge:	$I(D) \times I(D) \rightarrow I(D)$:	
	$merge([d_i, d_j), [d_p, d_q)) = [min(\{d_i, d_p\}), max(\{d_j, d_q\}))$, $j \ge p$ **and** $q \ge i$	
window:	$D \times I \times I \rightarrow I(D)$: $window(d_i, m, n) = [d_{i+m*n}, d_{i+m*(n+1)}), m > 0$	
windowno:	$D \times I \times D \rightarrow I$:	
	$windowno(d_i, m, d_j) = (j - i)\ div\ m, \quad j \ge i, m > 0$	
	$windowno(d_i, m, d_j) = -((i - j + m - 1)\ div\ m), \quad j < i, m > 0$	

APPENDIX B

VT-SQL SYNTAX

This appendix contains the syntax of VT-SQL. The syntax is slightly permissive, in that it allows the generation of certain constructs that are not legal. For example, the argument to a set function may not in turn be a set function and arithmetic operations between time-intervals are not allowed. The detailed restrictions which concern VT-SQL, have been given in the text under headings *General Rules* and *Additional Rules.* Note that minor syntax corrections may be made in the implementation.

1. DATA DEFINITION LANGUAGE

<table-definition> ::=
CREATE TABLE <table-name>
(<column-name-format-list>
[, **NORMALISED (<normalised-column-list>)]**
[, PRIMARY KEY (<key-column-list>]))

<column-name-format> ::= <column-name> <format>

<format> ::= <datatype> [NOT NULL [WITH | NOT DEFAULT] | WITH NULL]

<normalised-column> ::= <column-name>

<key-column> ::= <column-name> **[INTERVAL | POINT]**

<privilege-definition> ::= GRANT <privileges>
ON <table-name>
TO <grantees>

<privileges> ::= ALL PRIVILEGES
| <action-list>

<action> ::=

SELECT
| INSERT
| DELETE
| UPDATE (<column-name-list>)

<grantees> ::= PUBLIC
| <username-list>

<view-definition> ::=
CREATE VIEW <table-name> [(<column-name-list>)]
AS <query-spec>
[WITH CHECK OPTION]

2. DATA MANIPULATION LANGUAGE

<delete-stat> ::= DELETE FROM <table-name>
[PORTION <normalised-column-value-list>]
[<where-clause>]

<insert-stat> ::= INSERT [**<reformat-column-list>**]
INTO <table-name> [(<column-name-list>)]
<source-values>

<source-values> ::= VALUES (<values-list>)
| <query-spec>

<update-stat> ::= UPDATE <table-name>
PORTION <<normalised-column-value-list>]
SET <column-assignment-list>
<where-clause>

<column-assignment> ::= <column-name> = {<value-exp> | NULL}

<normalised-column-value> ::= **<normalised-column-name>** [**=<value-exp>**]

3. COMMON ELEMENTS

<all-set-function> ::= {AVG | MAX | MIN | SUM} [ALL] <value-exp>)
| { **COUNTDP** | **COUNTAP**} [ALL] <value-exp>)

<approx-num-literal> ::= <mantissa> E <exponent>

<boolean-factor> ::= [NOT] <boolean-primary>

<boolean-primary> ::= <predicate>
| (<search-condition>

<boolean-term> ::= <boolean-factor>
| <boolean-term> AND <boolean-factor>

<nonquote-char> ::= <digit> | <letter> | <special-char>

<char-represent> ::= <nonquote-char> | "

<char-string-literal> ::= '<char-represent>...'

<column-spec> ::= <column-name>
| <table-name>.<column-name>
| <correlation-name>.<column-name>
| <unsigned-integer>

<comparison-op> ::= = | <> | < | > | <= | >=
| **<interval-comparison-op>**

<data-type> ::= CHAR [(<length>)]
| VARCHAR [(<length>)]
| FLOAT4
| FLOAT8
| INTEGER1
| SMALLINT
| INTEGER4
| DATE
| **DATEINTERVAL**

<dateinterval-literal> ::= **'[<date>, <date>)'**

<distinct-set-function> ::= {AVG | MAX | MIN | SUM | COUNT}
(DISTINCT <column-spec>)
| { **COUNTDP** | **COUNTAP**} (DISTINCT <column-spec>)

<esc-char> ::= <value-spec>

<exact-num-literal> ::= [+ | -] <unsigned-int> [.<unsigned-int>]
| [+ | -] <unsigned-int>.
| [+ | -] .<unsigned-int>

<except-exp> ::= **<query-spec>**
EXCEPT [**<reformat-column-list>**]
<query-spec>

<exponent> ::= [+ | -] <digit>...

<factor> ::= [+ |-] <primary>

<from-clause> ::= FROM <table-ref-list>

<group-clause> ::= GROUP BY <column-spec-list>

<having-clause> ::= HAVING <search-condition>

<interval-comparison-op> ::= ***before***
| ***meets***
| ***loverlaps***
| ***lcovers***
| ***covers***
| ***rcovered***
| **=**
| ***rcovers***
| ***covered***
| ***lcovered***
| ***roverlaps***

| ***met***
| ***after***
| ***psubinterv***
| ***subinterv***
| ***psupinterv***
| ***supinterv***
| ***overlaps***
| ***merges***
| ***cp***
| ***precedes***
| ***follows***
| ***prequals***
| ***folequals***
| ***adjacent***

<length> ::= <unsigned-integer>

<literal> ::= <char-string-literal>
| <num-literal>
| **<dateinterval-literal>**

<mantissa> ::= <exact-num-literal>

<normalise-clause> ::= **NORMALISE ON <column-spec-list>**

<num-literal> ::= <exact-num-literal>
| <approx-num-literal>

<order-clause> ::= ORDER BY <order-item-list>

<order-column> ::= <column-spec>
| <unsigned-integer>

<order-item> ::= <order-column> [ASC | DESC]

<pattern> ::= <value-spec>

<predicate> ::=
<value-exp> <comparison-op> <value-exp>
| <value-exp> [NOT] BETWEEN <value-exp> AND <value-exp>
| <value-exp> [NOT] IN (<value-spec-list>)
| <column-spec> [NOT] LIKE <pattern> [ESCAPE <esc-char>]
| <column-spec> IS [NOT] NULL
| <value-exp> <comparison-op> <subquery>
| <value-exp> <comparison-op> ALL <subquery>
| <value-exp> <comparison-op> ANY <subquery>
| <value-exp> <comparison-op> SOME <subquery>
| <value-exp> [NOT] IN <subquery>
| EXISTS <subquery>

<primary> ::= <value-spec>
| <column-spec>
| <set-function-spec>
| <scalar-function>
| (<value-exp>)

<query-exp> ::= {<query-spec> | <union-exp> | **<except-exp>** }
[<order-clause>]

<query-spec> ::= SELECT [ALL | DISTINCT] <select-list>
<table-exp>

<reformat-clause> ::= **REFORMAT AS <reformat-item>**

<reformat-column> ::= <column-spec>
| <unsigned-integer>

<reformat-item> ::= **FOLD <column-spec-list>** [**<reformat-item>**]
| **UNFOLD** [**ALL**] **<column-spec-list>** [**<reformat-item>**]

<scalar-function> ::= <standard-SQL-scalar-function>
| ***now***
| ***start*** **(Dateinterval)**

| ***stop*** **(Dateinterval)**
| ***topoint*****(Dateinterval)**
| ***tointerv*****(Date)**
| ***interv*** **(Date, Date)**
| ***intervsect*** **(Dateinterval, Dateinterval)**
| ***succ*** **(Date, Integer)**
| ***dur*** **(Dateinterval)**
| ***span*** **(Date, Date)**
| ***dist*** **(Date, Date)**
| ***middle*** **(Dateinterval)**
| ***merge*** **(Dateinterval, Dateinterval)**
| ***window*** **(start-Date, Time-duration, Time-number)**
| ***windowno*** **(start-Date, Time-duration, Date)**

<search-condition> ::= <boolean-term>
| <search-condition> OR <boolean-term>

<select-list> ::= <value-exp-list> | *

<set-function-spec> ::= COUNT(*)
| <distinct-set-function>
| <all-set-function>

<subquery> ::= (SELECT {<value-exp> | *}
<table-exp>)

<table-exp> ::= <from-clause>
[<where-clause>]
[<group-clause>]
[<having-clause>]
[**<reformat-clause>**]
[**<normalise-clause>**]

<table-ref> ::= <table-name> [<correlation-name>]

<term> ::= <factor>
| <term> * <factor>

| <term> / <factor>

<union-exp> ::= <query-spec>
[UNION [{ ALL | **<reformat-column-list>** }]
<query-spec>

<values> ::= <literal> | NULL

<value-exp> ::= <term>
| <value-exp> + <term>
| <value-exp> - <term>

<value-spec> ::= <literal>
| <system-variable>

<where-clause> ::= WHERE <search-condition>

4. OTHER VT-SQL STATEMENTS

<index-definition> ::= CREATE [UNIQUE] INDEX <index-name>
ON <table-name>
(<index-column-list>)

<index-column> ::= <column-name> [ASC | DESC]

<drop-stat> ::= <drop-index-def>
| <drop-table-def>
| <drop-view-def>

<drop-index-def> ::= DROP INDEX <index-name>

<drop-table-def> ::= DROP TABLE <table-name>

<drop-view-def> ::= DROP VIEW <table-name>

APPENDIX C

EXAMPLES FROM THE TEST BED APPLICATION

In this appendix we demonstrate how VT-SQL can be used to answer queries of the test bed application. All the examples provided next, answer queries which are real in nature, not hypothetical. No queries have been identified which cannot be answered. In contrast, VT-SQL proves to be much more powerful.

Tables

We first introduce some tables which are used in the examples which follow. For each table we provide its key and a short description of its contents.

TRANSPLANTATION

Name	Date
John	d30
Peter	d40

Key: <*Name-i*>

The date on which a patient has had a transplant operation. For simplicity, we assume that each patient has only one transplantation.

INFECTION

Name	Cause	Time
John	Enterococcus s.p.	[d31, d34)
John	Enterococcus s.p.	[d40, d46)
John	Enterococcus s.p.	[d50, d60)
John	Proteus s.p.	[d33, d38)

Key: <*Name-i, Time-p, Cause-i*>

Time during which a patient was infected by some disease.

SURVIVAL

Name	Time
John	[d30, d50)
Peter	[d40, d45)

Key: <*Name-i*>

Survival time after transplantation.

DRUG

Name	Drug	Level	Time
John	Cyclosporine	121	[d30, d35)
John	Cyclosporine	58	[d35, d40)
John	Cyclosporine	110	[d40, d45)
John	Azathioprine	110	[d50, d60)

Key: <*Name-i, Time-p, Drug-i*>

Level of drug with which each patient was administered during each time-interval.

COMPLICATION

Name	Complication	Time
John	Hypotassemia	[d31, d40)
John	Hyperglycemia	[d33, d42)
John	Metabolic Alcolosis	[d37, d45)
John	Leukopenia	[d55, d60)

Key: <*Name-i, Time-p, Complication-i*>

Time during which a patient faced some complication.

CHOLESTEROL

Name	Level	Time
John	180	[d31, d33)
John	140	[d33, d36)
Peter	158	[d41, d44)
Peter	155	[d44, d46)
Peter	130	[d46, d50)

Key: <*Name-i, Time-p*>

The patients' cholesterol level for each date within the time-interval.

Queries

We notice that the key definition of the tables above justify the necessity of the use of the extended UNION, INSERT, DELETE, and UPDATE operations. For the retrieval queries we notice that some of them cannot be answered using SQL and some other can hardly be formulated

John was in addition infected by Proteus s.p. during [d38, d50). *Insert the data.*

INSERT
INTO INFECTION
VALUES ('John', 'Proteus s.p', '[d38, d50)')

INFECTION

Name	Cause	Time
John	Enterococcus s.p.	[d31, d34)
John	Enterococcus s.p.	[d40, d46)
John	Enterococcus s.p.	[d50, d60)
John	Proteus s.p.	[d33, d50)

Give the patients whose episodes of Hypotassemia had a duration of more than 5 days.

SELECT Name
FROM COMPLICATION
WHERE Complication = 'Hypotassemia'
AND *dur*(Time) > 5

GROUP BY Name

Give the patients who have been administered with Azathioprine during an episode of Leukopenia.

SELECT D.Name
FROM DRUG D, COMPLICATION C
WHERE D.Name = C.Name
AND D.Drug = 'Azathioprine'
AND C.Complication = 'Leukopenia'
AND D.Time *cp* C.Time
GROUP BY Name

Give John's levels of Cyclosporine from d30 to d40.

SELECT Level, *intervsect*(Time, '[d30, d41)')
FROM DRUG
WHERE Name = 'John'
AND Drug = 'Cyclosporine'
AND Time *cp* '[d30, d41)'

Give the time-intervals in which John had episodes of both Metabolic Alkolosis and Hyperglycemia and Hypotassemia

SELECT *intervsect*(C1.Time, *intervsect*(C2.Time, C3.Time))
FROM COMPLICATION C1, COMPLICATION C2,
COMPLICATION C3
WHERE C1.Name = 'John'
AND C1.Name = C2.Name
AND C2.Name = C3.Name
AND C1.Time *cp* C2.Time
AND C2.Time *cp* C3.Time
AND C3.Time *cp* C1.Time
AND C1.Complication = 'Metabolic Alkolosis'
AND C2.Complication = 'Hyperglycemia'
AND C3.Complication = 'Hypotassemia'

For each patient give any complications he has had 10 days after the transplant operation.

```
SELECT      C.Name, C.Complication
FROM        TRANSPLANTATION T, COMPLICATION C
WHERE       T.Name = C.Name
AND         tointerv(succ(Date, 10)) cp Time
GROUP BY    Name, Complication
```

Give the patients who have had some infection during the first thirty days after the transplant operation.

```
SELECT      T.Name
FROM        TRANSPLANTATION T, INFECTION I
WHERE       T.Name = I.Name
AND         window(Date, 30, 0) cp Time
GROUP BY    Name
```

Give the level of Cyclosporine with which all patients were administered in the third week after the transplant operation.

```
SELECT D.Name, D.Level
FROM   TRANSPLANTATION T, DRUG D
WHERE  T.Name = D.Name
AND    window(Date, 7, 3) cp Time
```

Give the number of patients who have survived for more than three months after the transplantation.

```
SELECT count(*)
FROM   SURVIVAL S, TRANSPLANTATION T
WHERE  S.Name = T.Name
AND    windwono(Date, 30, stop(Time)) >= 2
```

Give John's levels of Cyclosporine for each of the days in [d30 to d41).

```
SELECT      Level, intervsect(Time, '[d30, d41)')
FROM        DRUG
WHERE       Name =  'John'
AND         Drug =  'Cyclosporine'
AND         Time cp '[d30, d41)'
```

```
REFORMAT AS
        UNFOLD Time
```

Give the greatest time-intervals during which John was administered with Cyclosporine.

```
SELECT      Time
FROM        DRUG
WHERE       Name = 'John'
AND         Drug = 'Cyclosporine'
REFORMAT AS
        FOLD Time
```

Give the number of days John has been administered with Cyclosporine from d30 to d38.

```
SELECT countdp(intervsect(Time, '[d30, d39)'))
FROM   DRUG
WHERE  Drug =  'Cyclosporine'
AND    Name =  'John'
AND    Time cp '[d30, d39)'
```

Give the number of days John was administered with Cyclosporine from the transplant date to the present.

```
SELECT countdp(intervsect(Time, interv(Date, 'now')))
FROM   DRUG D, TRANSPLANTATION T
WHERE  T.Name =  'John'
AND    T.Name =  D.Name
AND    Drug   =  'Cyclosporine'
AND    Time   cp interv(Date, 'now')
```

Give the level of cholesterol while John had an Enterococcus s.p infection.

```
SELECT  Level, intervsect(C.Time, I.Time)
FROM    CHOLESTEROL C, INFECTION I
WHERE   C.Name =  'John'
AND     C.Name =  I.Name
AND     C.Time cp I.Time
```

Give the patients whose cholesterol level surpassed 150 at some time of the during the first 30 days after the transplant operation.

```
SELECT T.Name
FROM   CHOLESTEROL C, TRANSPLANTATION T
WHERE  C.Name  = T.Name
AND    C.Level > 150
AND    window(T.Date, 30, 0) cp C.Time
```

For each patient give the cholesterol level in the third week after the transplant operation.

```
SELECT C.Name, C.Level, intervsect(C.Time, window(T.Date, 7, 3))
FROM   CHOLESTEROL C, TRANSPLANTATION T
WHERE  C.Name =  T.Name
AND    C.Time cp window(T.Date, 7, 3)
```

Give the week number after transplant operation that John's cholesterol level became less than 140.

```
SELECT windowno(T.Date, 7, min(start(C.Time)))
FROM   TRANSPLANTATION T, CHOLESTEROL C
WHERE  T.Name  = 'John'
AND    T.Name  = C.Name
AND    C.Level < 140
```

For each patient give the time elapsed between two episodes of Enterococcus s.p.

```
SELECT I1.Name, dist(start(I2.time), stop(I1.Time))
FROM   INFECTION I1, INFECTION I2
WHERE  I1.Cause  = 'Enterococcus s.p.'
AND    I1.Name   = I2.Name
AND    I1.Cause  = I2.Cause
AND    stop(I1.Time) < start(I2.Time)
AND    stop(I1.Time) =  ( SELECT max(stop(I3.Time))
                          FROM   INFECTION I3
                          WHERE  stop(I3.Time) < start(I2.Time)
                          AND    I3.Cause = 'Enterococcus s.p.'
                          AND    I3.Name  = I1.Name)
```

REFERENCES

[01P 93] 01 PLIROFORIKI 'Specification of valid time formalism', ORES Deliverable C3, Athens, 1993.

[CPH 93] CPH 'User Requirements', ORES Deliverable B2, Madrid, 1993.

[Date 86] C. J. Date. 'A guide to the SQL standard', 2nd Edition, Addison-Wesley, 1982.

[INGRES 89] Ingres 'SQL reference manual', Release 6.3, VAX/VMS, 1989.

[Lans 88a] R. F. van der Lans. 'Introduction to SQL', Addison-Wesley, 1988.

[Lans 88b] R. F. van der Lans. 'The SQL standard. A complete reference', Prentice Hall, 1988.

[Lorentzos et al 92] N. A. Lorentzos, A. Poulovasilis and C. Small. 'Optimized update operations for multi-dimensional interval data', Int. Rept., Informatics Laboratory, Agricultural University of Athens, 1993.

[Lorentzos 93] N. A. Lorentzos. 'Properties of functional dependencies', Int. Rept., Informatics Laboratory, Agricultural University of Athens, 1993.

[Navathe & Ahmed 86] S. B. Navathe, and R. Ahmed. 'A temporal relational model and a query language', Tech. Rept. TR-85-16, Department of Computer and Information Sciences, University of Florida, 1986.

[Sarda 90] N. L. Sarda. 'Extensions to SQL for historical databases', IEEE Transactions on Knowledge and Data Engineering, Volume 2, Number 2, 1990, pp 220-230.