ESPRIT III

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

DELIVERABLE D4.1
IMPLEMENTATION OF VALID TIME SQL

Responsible: 01 PLIROFORIKI
University of Athens
Agricultural University of Athens
INFORMATION DYNAMICS

Availability: Public

April 1994

IMPLEMENTATION OF VALID TIME SQL

(DELIVERABLE D4)

Responsible: **01 PLIROFORIKI**
University of Athens
Agricultural University of Athens
INFORMATION DYNAMICS

ABSTRACT

This report concerns the development of Valid Time SQL. The development has been based on previous ORES deliverables, concerning the Specification and design of VT-SQL. Deviations undertaken during the implementation phase are also reported.

TABLE OF CONTENTS

1. INTRODUCTION

This deliverable concerns the development of valid time SQL (VT-SQL). The development is in accordance with the specification of VT-SQL [01 P 93d]. Deviations either from the specifications of V-SQL [01 P 93d] or the design of VT-SQL [01 P 93c], are also reported. It should e noted that some optimisation, has in addition, been incorporated. The remainder of this report is outlined as follows:

In section 2 we describe the instalation of VT-SQL. Section 3 is the user's guide. In section 4 we report on the implementation of VT-SQL Conclusions are drawn in the last section. Finally, in Appendix A we give a list of the tests for the evaluation of VT-SQL

2.VT-SQL INSTALLATION

In order to install the VT-SQL query processor and create the VT-SQL system tables, the following procedure must be carried out:

1. Log in as an authorised INGRES user. If you are not sure whether such a user exists, log in as the *ingres* user.

2. Change your working directory to the directory where you want to install the VT-SQL query processor. Make sure that you have write permission on that directory. If you have logged in as the *ingres* user, installing the query processor in the directory *`~/bin`* or in the directory *`~/utility`* will make it automatically available to all INGRES users. (The tilde character (~) is the C-Shell (`csh`) shorthand for your home directory. If you are using Bourne shell (`sh`), substitute the tilde character with the notation `$HOME`.)

3. Insert the cartridge into the appropriate device. Enter the command

   ```
   tar xvf deviceName
   ```

   where *deviceName* is the name of the special file that corresponds to the device, e.g. */dev/rst0*. The VT-SQL query processor will be extracted in a file named *VT-SQL*. as well as the script for the creation of the VT-SQL system tables named *cr_vtsql_systables.*

4. Ensure that you have execute permission on the VT-SQL command processor and the above script, by issuing the command

   ```
   chmod +x VT-SQL cr_vtsql_systables
   ```

5. Create the system tables for a database under the name <DatabaseName> by issuing the command

   ```
   sql <DatabaseName> < cr_vtsql_systables
   ```

The VT-SQL query processor is now installed and the system tables created. *(Note that the routines of Vlid time Relational Algebra are also installed in this way).

3. USER'S GUIDE

3.1 USING THE VT-SQL QUERY PROCESSOR

Invoking the VT-SQL query processor

Once installed, the VT-SQL query processor may be invoked by entering the command

```
VT-SQL databaseName [filename]
```

where *databaseName* is the name of the INGRES database that the user issuing the command wants to work with. The user must be authorised to use the specified database. If the optional *filename* parameter is omitted, the VT-SQL query processor is started in interactive mode, accepting input from the keyboard and displaying results and error messages to the screen. If the *filename* parameter is specified, the VT-SQL query processor is started in batch mode, reading commands from file *filename*. Results and error messages are still printed on the screen, but this can be changed using the standard UNIX redirection notations. Note that input redirection and specification of the *filename* parameter are not equivalent: If input is redirected, interactive mode semantics apply to the session, while specification of the *filename* parameter implies batch mode semantics for the session. Thus, the command

```
VT-SQL database filename
```

reads input from file *filename* with batch mode semantics, while the command

```
VT-SQL database < filename
```

reads input from the same file with interactive mode semantics.

Comments

1. INGRES servers must be running when the user invokes the VT-SQL query processor, or the program will halt, issuing an appropriate error message.

2. If the specified database does not exist, the program will notify the user and halt.

3. The user is subject to access restrictions imposed by the database administrator and the table owners.

4. The optional *filename* parameter, if specified, must identify an existing file, on which the user has read permission.

Working with the VT-SQL query processor

The VT-SQL query processor accepts input either from the keyboard or from a file, depending on the invocation syntax. Input fed to the VT-SQL query processor must comply to the syntax rules which are specified in [1], and summarised below. The VT-SQL query processor also accepts input conferment to the Extended Syntax Specifications, which are described below.

In the subsequent paragraphs, the following notations will be used:

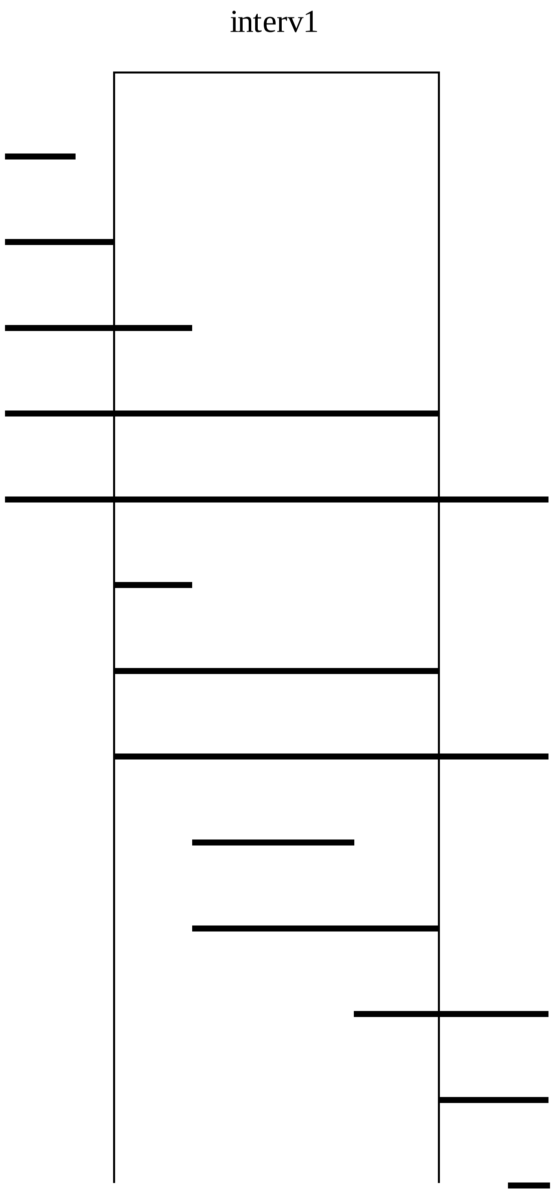
- Normal writing indicates reserved words, which must be typed as they appear in the document.
- Italics indicate that the term should be substituted by an appropriate string.
- Terms enclosed in brackets ([ ]) are optional.
- Terms enclosed in braces ({ }) are optional and may be repeated several times.
- term1 | term2 means "either term1 or term2"
- Parentheses are used to group terms. Parentheses that must be typed literally, are enclosed in single quotes (' ')

3.2 VT-SQL DATA TYPES

VT-SQL supports all the data types supported by INGRES. Additionally, in order to support valid time data, a new data type, namely DATEINTERVAL, has been introduced for the representation of time intervals. For every DATEINTERVAL we use the notation [di,dj), where di, dj of type DATE and $d_i < d_j$.

3.3 VT-SQL RELATIONAL OPERATORS

The VT-SQL query processor supports all standard SQL operators. In addition, predicates for dateinterval comparisons are supported, that may be used to form conditions in the **where** and **having** clauses. The predicates and the cases for which each one is true are illustrated in Figure 1.



1.	interv1	before	interv2
2.	interv1	meets	interv2
3.	interv1	loverlaps	interv2
4.	interv1	lcovers	interv2
5.	interv1	covers	interv2
6.	interv1	rcovered	interv2
7.	interv1	=	interv2
8.	interv1	rcovers	interv2
9.	interv1	covered	interv2
10.	interv1	lcovered	interv2
11.	interv1	roverlaps	interv2
12.	interv1	met	interv2
13.	interv1	after	interv2

14.	interv1	psubinterv	interv2	This predicate is true when `interv1` is a pure subinterval of `interv2` (one of the conditions 6, 9 and 10 is true).
15.	interv1	subinterv	interv2	This predicate is true when `interv1` is a subinterval of `interv2` (one of the conditions 6, 9, 7 and 10 is true).
16.	interv1	psupinterv	interv2	This predicate is true when `interv1` is a pure superinterval of `interv2` (one of the conditions 4, 5 and 8 is true).

17.	interv1	supinterv	interv2	This predicate is true when *interv1* is a superinterval of *interv2* (one of the conditions 4, 5, 7 and 8 is true).
18.	interv1	cp	interv2	This predicate is true when the two intervals have common points (one of the conditions 3, 4, 5, 6, 7, 8, 9, 10 and 11 is true).
19.	interv1	adjacent	interv2	This predicate is true when the starting date of *interv1* is the successor of the ending point of *interv2* or the starting date of *interv2* is the successor of the ending point of *interv1* (one of the conditions 2 and 12 is true).
20.	interv1	overlaps	interv2	In other words, the predicate is true when *interv1* and *interv2* have common points, but neither *interv1* is a subinterval of *interv2*, nor *interv2* is a subinterval of *interv1* (one of the conditions 3 and 11 is true).
21.	interv1	merges	interv2	This predicate is true when *interv1* and *interv2* have common points, or when *interv1* and *interv2* are adjacent (one of the conditions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 and 12 are true).
22.	interv1	prequals	interv2	This predicate is true when *interv1* starts before or at the same date as *interv2* and ends before or at the same date as *interv2* (one of the conditions 1, 2, 3, 4, 6 and 7 are true).

23.	interv1	preceds	interv2	This predicate is equivalent to the *prequals* predicate, except for the case of operand equality, in which the *preceds* predicate yields false, whereas the *prequals* predicate yields true (one of the conditions 1, 2, 3, 4 and 6 are true).
24.	interv1	folequals	interv2	This predicate is true when *interv1* starts after or at the same date as *interv2* and ends after or at the same date as *interv2* (one of the conditions 7, 8, 10, 11, 12 and 13 are true).
25.	interv1	follows	interv2	This predicate is equivalent to the *folequals* predicate, except for the case of operand equality, in which the *follows* predicate yields false, whereas the *folequals* predicate yields true (one of the conditions are 8, 10, 11, 12 and 13 true).

3.4 VT-SQL FUNCTIONS

The VT-SQL query processor supports the following functions, in addition to the functions provided by INGRES.

1. countap(dateint)

countap is an aggregate function, accepting an argument of type DATEINTERVAL. The function operates on the collection of values specified by its argument and returns the number of time points in all intervals. The *countap* function does not accept the *all* and *distinct* qualifiers.

Examples

The following statement finds the total number of days that certain patient has hospitalised.

```
select countap(hosp_stay_int)
from patient
where medrec_no = 'ABCDEF'
```

2. **ctointerv(string)**

   ctointerv accepts an argument of the form *'date1,date2'*. *date1* and *date2* should be valid dates and also *date1* < *date2*. The function operates on a string, checks if it represents a valid dateinterval and returns a DATEINTERVAL.

   Examples

   ctointerv('1990-01-01,1991-01-01') yields [1990-01-01, 1991-01-01)

   ctointerv('1991-01-01,1990-01-01') returns an error message

3. **dist(date1,date2)**

   Function *dist* accepts two arguments of type DATE and returns the number of days between its two arguments. The result is of type INTEGER and is always non-negative.

   Examples

   dist('1990-01-01', '1990-02-01')
   yields 31.

4. **dur(dateint)**

   Function *dur* accepts an argument of type DATEINTERVAL and returns the number of time points included in its argument. The result is of type INTEGER.

   Examples

   dur('[1990-01-01, 1991-01-01)')
   yields 365.

5. **interv(date1, date2)**

   Function *interv* accepts two arguments of type DATE. Its result is a DATEINTERVAL, starting at *expr1* and ending at *expr2*. An error occurs if *expr1* is after *expr2*.

   Examples

   interv('1990-01-01', '1991-01-01') yields [1990-01-01, 1991-01-01).

interv('1991-01-01', '1990-01-01') results to an error.

6. intervsect(dateint1, dateint2)

Function *intervsect* accepts two argument of type DATEINTERVAL and returns the common points of its argument. The result is of type DATEINTERVAL. An error occurs if the two arguments do not have any common points.

Examples

intervsect('[1990-01-01, 1991-01-01)', '[1990-02-01, 1993-01-01)')
yields [1990-02-01, 1991-01-01).

intervsect('[1990-01-01, 1991-01-01)', '[1989-02-01, 1993-01-01)')
yields [1990-01-01, 1991-01-01).

intervsect('[1990-01-01, 1991-01-01)', '[1992-01-01, 1993-01-01)')
results to an error (the two arguments do not have any common points).

7. maxdate()

This function returns the maximum date supported. It takes no arguments and yields a result of type DATE. The maximum date currently supported by the system is '2030-12-30'.

Examples

maxdate() yields '2030-12-30'

8. merge(dateint1, dateint2)

Function *merge* accepts two arguments of type DATEINTERVAL and returns a DATEINTERVAL containing all the time points in both arguments. An error occurs if the two arguments cannot be merged, i.e. one of the conditions *dateint1 before dateint2* and *dateint1 after dateint2* is true.

Examples

merge('[1990-01-01, 1991-01-01)', '[1990-06-01, 1992-01-01)')
yields [1990-01-01, 1992-01-01).

merge('[1990-01-01, 1991-01-01)', '[1990-06-01, 1990-08-01)') yields [1990-01-01, 1991-01-01).

merge('[1990-01-01, 1991-01-01)', '[1991-01-01, 1992-01-01)')
yields [1990-01-01, 1992-01-01).

merge('[1990-01-01, 1991-01-01)', '[1992-06-01, 1990-08-01)')
results to an error, because the two arguments are neither overlapping nor adjacent.

9. middle(dateint)

Function *middle* accepts an argument of type DATEINTERVAL and returns the midpoint of its argument. The result is of type DATE.

Examples

middle('[1990-01-01, 1990-01-03)') yields '1990-01-02'.

10.mindate()

This function returns the minimum date supported. It takes no arguments and yields a result of type DATE. The maximum date currently supported by the system is '1970-01-01'.

Examples

mindate() yields '1970-01-01'

11.now()

This function returns the current date. It takes no arguments and yields a result of type DATE.

12.span(date1, date2)

This function accepts two arguments of type DATE and returns the number of days between its two arguments. The result is of type INTEGER and may be positive, zero or negative depending on the relative position of *date1* and *date2*.

Examples

span('1990-01-01', '1989-01-01') yields 365 span('1990-01-01', '1990-02-01') yields -31

13.start(dateint)

Function *start* takes an argument of type DATEINTERVAL and returns the starting date of its argument. The result is of type DATE.

Examples

start('[1990-01-01, 1990-01-03)') yields '1990-01-01'

14.stop(dateint)

Function *start* takes an argument of type DATEINTERVAL and returns the starting date of its argument. The result is of type DATE.

Examples

stop('[1990-01-01, 1990-01-03)') yields '1990-01-03'

15.succ(date1, int1)

Function succ accepts one argument of type DATE and one argument of type INTEGER. The result of the function is a date *int1* days after *date1*.

Examples

stop('1990-01-01', 2) yields '1990-01-03'

16.tointerv(date1)

Function *tointerv* accepts an argument of type DATE and returns the trivial DATEINTERVAL which includes only the date specified by the argument.

Examples

tointerv('1990-01-01') yields '[1990-01-01, 1990-01-02)'

17.topoint(dateint)

Function *topoint* accepts an argument of type DATEINTERVAL. If the argument is a trivial dateinterval (i.e. a DATEINTERVAL including a single time point), the function returns the start of its argument. If the argument is not a trivial DATEINTERVAL, an error occurs.

Examples

topoint('[1990-01-01, 1990-01-02)') yields '1990-01-03'.

topoint('[1990-01-01, 1990-01-03)') results to an error.

18.window(start_date, period_length, num_periods)

Function window accepts three arguments. The first argument specifies a starting date, the second a period length (number of days), and the third a number of periods. If the *num_periods* argument is positive, the function returns a DATEINTERVAL, starting *num_periods* periods (i.e. *period_length * (num_periods - 1)* days) after *start_date* and ending *num_periods + 1* periods after *start_period*. If the *num_periods* argument is negative, the function returns a DATEINTERVAL starting *num_periods* periods (i.e. *period_length * num_periods* days*)* before *start_date*. The duration of the result dateinterval is *period_length* days. Argument *period_length* must be positive, and argument *num_periods* must be non-zero.

Examples

window('1990-01-01', 10, 2) yields [1990-01-11, 1990-01-21).

window('1990-01-01', 10, -3) yields [1989-12-02, 1989-12-12).

19.windowno(start_date, period_length, end_date)

Function *windowno* accepts three arguments. The first argument specifies a starting date, the second a period length (number of days), and the third an ending date. The result is the number of periods (each one lasting *period_length* days) between *start_date* and *end_date*, and will be negative if *end_date* is before *start_date*. The result, however, cannot be zero.

Examples

windowno('1990-01-01', 10, '1990-01-20') yields 2.

windowno('1990-01-01', 10, '1989-12-12') yields -2.

3.5 VT-SQL STATEMENTS

CREATE INDEX

Syntax

CREATE [UNIQUE] INDEX *indexName* ON *tableName*
'('*columnName* {, *columnName*}')'

Description

The CREATE INDEX statement is used to build indexes on tables, allowing for rapid data retrieval. The **unique qualifier**, if specified, ensures that no two rows of table *tableName* may contain identical data in the columns on which the index is created. However, if the table already contains such data, the CREATE INDEX statement will fail.

Comments

- The CREATE INDEX statement will also fail if an object (table or index) with the name *indexName* already exists in the database.
- Names starting with `ii` are reserved by INGRES, and the VT-SQL query processor reserves names starting with `jj` and `tt`, as well as the names `VT-SQL_keys` and `VT-SQL_norms`. Reserved names should not be used as index names in a CREATE INDEX statement.

Examples

1. The following statement creates a unique index on the `medrec_no` column of table `patient`.

   ```
   create unique index index1 on patient(medrec_no)
   ```

2. The following statement creates a non-unique index on columns `type` and `cause` of table `operation`.

   ```
   create index index2 on operation(type, cause)
   ```

CREATE TABLE

Syntax

create table *tableName*
(*columnName dataType* [not null] {, *columnName dataType* [not null]}
[normalised (*columnName*)]
[primary key (*columnName* [interval | point] {, *columnName* [interval | point]})])

Description

VT-SQL CREATE TABLE statement is an extended version of the standard SQL corresponding statement. Its enhancements reside in the provision of the new data type, namely DATEINTERVAL, and the ability to specify the primary keys and normalisation columns of the table to be created. Supported datatypes are the standard SQL datatypes, as well as DATEINTERVAL datatype.

In order to ensure the efficient use of CREATE TABLE statement, a number of limitations should be considered when using this statement. Normalisation attributes should be present in the table column list and not be duplicated. All columns present in the NORMALISED clause must be of type DATEINTERVAL. Such a column should also be present in the PRIMARY KEY section, if there is any, and -moreover- be of type POINT.

Primary key attributes must be present in the table column list, whereas declaration of duplicate primary keys is illegal. Columns participating in the PRIMARY KEY section as POINT should also be present in the NORMALISED clause.

Columns declared as primary keys should be either of type DATEINTERVAL or POINT, following the limitations imposed above. If no type is specified, the default type, namely DATEINTERVAL, is used. Every column present in either of the statement extended clauses should acquire the *not null* property, that is, whether or not it is declared as nullable, it is ultimately created as *not null*.

Comments

The CREATE TABLE statement will fail if an object (table or index) with the name *tableName* already exists in the database. Names starting with `ii` are reserved by INGRES, and the VT-SQL query processor reserves names starting with `jj` and `tt`, as well as the names `vtsql_keys` and `vtsql_norms`. Reserved names should not be used as table names in a CREATE TABLE statement.

Examples

1. Create table *inflation* with fields *Country* (character 15), *Percetage* (real) and *Time* (dateinterval), with primary key on *Country* and *Time*.

```
create table inflation (Country char(15),
        Percentage real,
        Time dateinterval
        Primary Key (Country interval, Time interval)
```

2. Create table salary with fields Name (character 15), Time (dateinterval) and *Amount* (integer) , normalised on *Time*, with primary key on *Name* and *Time*.

```
create table salary (Name char(15),
        Time dateinterval,
        Amount integer
        Normalised (Time)
```

3. Create table salary with fields Name (character 15), Time (dateinterval) and *Amount* (integer) , normalised on *Time*, with primary key on *Name* and *Time*.

```
create table salary (Name char(15),
        Time dateinterval,
        Amount integer
        Normalised (Time)
        Primary Key (Name interval, Time point))
```

4. Create table shift with fields Name (character 15), Time (dateinterval) and Date (dateiterval), normalised on *Time*, with primary key on *Date* and *Time*.

```
create table shift (Name char(15),
        Time dateinterval,
        Date dateinterval
        Normalised (Time)
        Primary Key (Date interval, Time point))
```

DELETE

Syntax

```
delete from tableName
[portion columnName = period]
[where condition]
```

Description

The **delete** statement removes rows' portions from a specified table that satisfy the *condition* in the **where** clause and they match with the *period* in the **portion** clause. If both the **where** and **portion** clauses are omitted, the statement deletes all rows in the table. The result is a valid but empty table. If the **where** clause is omitted, but there does exist a **portion** clause, the statement will delete the portions' of those rows that match with the *period* in **portion** clause. If only a **where** clause exists, the statement will remove the rows that satisfy the *condition* in the **where** clause.

The *columnName* stated in the **portion** clause must refer to the column that the table is normalised on, and *period* must be an expression yielding a result of type **interval**. If table *tableName* is not normalised on some column, the **portion** clause should not be specified. The **portion** clause restricts the scope of the **delete** statement to the time points included in result of *period*.

The *condition* in the -optional- **where** clause may be any valid condition, as described in the paragraphs referring to the **select** statement.

Comments

- If table *tableName* is normalised on some column, and the **portion** clause is specified, then the execution of a **delete** statement may not decrease the number of rows in the table. The number of rows may increase, decrease or remain the same, depending on the row foldings that will take place.

Examples

1. Delete all employees whose salary is over 1000000

```
delete from employee
where salary > 1000000
```

2. Delete all employees data for the period '[1990-01-02, 1990-03-03)'

```
delete from employee
portion time = '[1990-01-02, 1990-03-03)'
```

3. Delete John's data for the period '[1990-01-02, 1990-03-03)'

```
delete from employee
portion time = '[1990-01-02, 1990-03-03)'
where name = 'John'
```

4. Delete all employees

```
delete from employee
```

DROP INDEX

Syntax

DROP INDEX *indexName*

Description

The DROP INDEX statement destroys an index created via the CREATE INDEX statement.

Comments

- The user should not use the DROP INDEX statement in order to remove indexes whose names start with `ii`, `jj` or `tt`. These indexes are reserved by the INGRES DBMS and the VT-SQL query processor.

Examples.

1. The following statement destroys the index *`index1`*.

   ```
   drop index index1
   ```

DROP TABLE

Syntax

drop table *tableName*

Description

The DROP TABLE statement removes table ***tableName*** from the database, and eliminates any data concerning this table, including indexes and primary key information. The user should not use the DROP TABLE statement in order to remove tables `vtsql_keys`, `vtsql_norms` as well as tables whose names start with `ii`, `jj` or `tt`.

When executing the VT-SQL DROP TABLE statement all entries referring to table *table-name* are removed from the systems tables. The table itself no longer exists in the database. The DBMS also takes care of all structures, as indexes, relating to this table. Consequently, all indexes previously declared are therefore removed from the Data Base without any further actions required.

Comments

No actions have to be taken for removing previously defined indexes based on table table-name.

Examples

1. The following statement destroys the table *salary*.

   drop table salary

HELP

Syntax

HELP [*object_name*]

Description

The HELP command provides a list of the tables which the user is allowed to access in the current database, or information about a particular database object. If the parameter *object_name* is not specified, then a table list is displayed. If, however, the parameter is specified, it must designate an existing table or index owned by the user issuing the command or the *ingres* user. Information displayed about a table includes the table schema as well as primary keys and normalised columns defined for that table.

Examples

1. The following statement displays a list of the tables in the current database on which the user has access.

   ```
   help
   ```

1. The following statement displays information about table *patient*.

   ```
   help patient
   ```

INSERT

Syntax

```
insert into tableName [(column-list)]
values (value-or-null-list)

or

insert into tableName [(column-list)] query
```

Description

VT-SQL INSERT statement is an extended version of the standard SQL corresponding statement. Its enhancements reside in the use of the NORMALISE ON and REFORMAT AS sections that may be present in the subselect clause, as well as the functionality originating from the definition of the table and concerning its primary keys and normalisation columns.

Though the use of the INSERT statement the user inserts values in the insertion table. Values may be declared in two ways:

- as single values (forming a row, or sometimes a subset of a row).
- as the result of a subselect on any table.

It is evident that in both cases the insertion values must comply with the insertion table schema. It is also possible to insert values in specific columns of the insertion table, declared in the column-list clause. In the second case, attention should be paid to the columns of the table acquiring the not null property. If no value is specified for these columns it is unavoidable to receive an error message.

Handling of the subselect clause is much alike the SQL subselect. The extra functionality reside in the NORMALISE ON and REFORMAT AS sections, through which we can specify the normalisation and reformatting forms to be applied on the intermediate table that holds the results of the conventional subselect. The intermediate table is reformed in order to acquire the insertion table schema.

The columns specified in the NORMALISE ON and REFORMAT AS sections may have the standard SQL format. These clauses are viewed in a similar manner to the GROUP BY clause; thus, all limitations for the GROUP BY are depicted for NORMALISE ON and REFORMAT AS. There is also the possibility to use the number representing the order of the column in the resulting table instead of using any of the conventional formats. This is analogous to the ability provided within the ORDER BY clause in the standard SQL.

After the execution of the extended subselect statement, all results acquired are to be inserted in the insertion table. In case this table is not normalised, the values are simply inserted thus completing the whole operation. Whether the table is normalised or not, is specified in the CREATE TABLE statement given for this table. Normalisation attributes reside in the system table vtsql_norms and may be viewed through a single select query. In case the table is normalised, an extensive checking is performed in order to ensure that the values to be inserted do not violate the insertion table primary keys. Primary keys are specified in the CREATE TABLE statement and reside in the corresponding system table, namely vtsql_keys. Any values encountered to cause duplication, result in the abortion of the whole operation, and the display of an appropriate message.

In case that no duplication is encountered, the values are inserted in the insertion table. The table is consequently normalised according to the attributes mentioned above.

Comments

- If table *tableName* is normalised on some column, then the execution of an INSERT statement may not increase the number of rows in the table. The number of rows may increase, decrease or remain the same.
- The query in the second form of the INSERT statement must be a single select clause, that is it may not consist of two select clauses combined by the **union**, **union all** or **except** operators.
- Insertion attempts should not be made for the VT-SQL system tables, namely vtsql_norms and vtsql_keys

Examples

1. Insert into table salary the row (John, 1000,'[1991-01-01, 1993-01-01)').

```
insert
into salary
values  ('John', 1000, '[1991-01-01, 1993-01-01)')
```

2. Insert into table inflation the row (A, '[1990-01-02, 1990-03-03)').

```
insert
into inflaction(Country, Time)
values ('A', '[1990-01-02, 1990-03-03)')
```

3. Insert into table salary all data related to John from table h_salary.

```
insert
into salary
    select Name, Amount, Time
    from h_salary
    where Name='John'
```

4. Insert into table patient all drugs and the corresponding date intervals for which they were given to patient John.

```
insert
into patient
    select Name, Drug, Time
    from drug
    where Name='John'
    normalised on Time
```

SELECT

Syntax

```
SELECT [DISTINCT | ALL] target-column-list
FROM table-list
[WHERE condition]
[GROUP BY column-list [HAVING condition]]
[REFORMAT AS
     (FOLD | UNFOLD | UNFOLD ALL) result-column-list
     {(FOLD | UNFOLD | UNFOLD ALL) result-column-list}]
[NORMALISE ON result-column-list]
[(UNION | UNION ALL | EXCEPT) [result-column-list]
SELECT [DISTINCT | ALL] target-column-list
FROM table-list
[WHERE condition]
[GROUP BY column-list [HAVING condition]]
[REFORMAT AS
     (FOLD | UNFOLD | UNFOLD ALL) result-column-list
     {(FOLD | UNFOLD | UNFOLD ALL) result-column-list}]
[NORMALISE ON result-column-list] ]
[ORDER BY result-column-list]
```

Description

The SELECT statement retrieves data from tables in the database and displays it on the screen. The SELECT statement has been extended, compared to the standard SQL SELECT statement, allowing for reformatting and normalisation of target columns, application of the VT-Algebra operations PUNION and PEXCEPT to query results and usage of all predicates and functions involving time points and dateintervals.

The *target-column-list* is a set of comma-separated select items or an asterisk. Each select item may be a column name (qualified or non-qualified[1]) or an expression. The default name for each select item is the column name, if the select item is a column name, or a name of the form `colI`, if the select item is an expression (`I` is an integer, increasing for every expression encountered; the leftmost expression is named `col1`). The default name can be changed to *newName*, using either the `newName = select item` or the `select item as newName` notation. If the *target-column-list* is an asterisk, all columns belonging to the tables in *table-list* are retrieved.

The *table-list* is a comma-separated list of table names from which the SELECT statement will retrieve data. Standard SQL table aliasing (`tableName aliasName`) is supported.

The *condition* in the **where** and **having** clauses is any valid Boolean expression, and may contain simple column and value comparisons, standard INGRES datatype functions, functions and predicates involving dateintervals and time points (dates), and subqueries. Simple conditions may be combined to more complex ones, using the **and** and **or** Boolean operators. The **not** operator may also be used to negate the meaning of a condition. All standard SQL subquery types are supported. For dateinterval datatypes, the predicates `precedes`, `prequals`, `equals`, `follows` and `folequals` may be used to form subqueries, e.g. the condition:

treatmentPeriod follows (select infectionPeriod from infected)

is valid. Standard SQL qualifiers **all** and **any** and the **not** operator may be used for subqueries, in conjunction with the permitted dateinterval predicates, thus the conditions

treatmentPeriod follows all (select infectionPeriod from infected)

and

treatmentPeriod not follows any (select infectionPeriod from infected)

[1]A qualified column name has the form `tableName.columnName`, and refers to column `columnName` of table `tableName`. Non-qualified column names consist of the `columnName` part only.

are valid. The first condition is true if the value of `treatmentPeriod` overlaps with the ending part of every value of `infectionPeriod` retrieved by the subquery, while the second condition is true if there exists a value of `infectionPeriod` retrieved by the subquery, such that its ending part overlaps with the value of `treatmentPeriod`. None of the dateinterval predicates which are not listed above is allowed to be used to form subqueries. The standard SQL relational operators **>**, **>=**, **<** and **<=** are supported for dateintervals, and correspond to the `precedes`, `prequals`, `equals`, `follows` and `folequals` predicates, respectively.

The *column-list* in the **group by** clause is a list of comma-separated column names, which may be qualified or non-qualified.

The *result-column-list* in the **reformat**, **normalise** and **order by** clauses is a list of comma-separated target column specifiers. Each target column specifier may be a column name (qualified or non-qualified), or an integer, indicating the position of the desired column in the *target-column-list*. The same rules apply to the *result-column-list* that may be specified after the **union**, **union all** and **except** keywords, which may be used to combine query results.

If either *`union`*, *`union all`* or *`except`* keyword is used to combine query results, then the results of the two queries must be union-compatible, which means that:

1. the number of columns of the two results must be the same

2. the corresponding columns must be type-compatible, as type compatibility is defined by INGRES.

If the *`union`*, *`union all`* or *`except`* keyword is not followed by a *result-column-list*, the corresponding standard relational algebra operator is applied to the results of the two queries, in order to produce the final result. If, however, a *result-column-list* is present, the valid time algebra operators **PUNION** and **PEXCEPT** are used instead. The columns specified in the *result-column-list* must be of type *`dateinterval`* or *`date`*. If column names are used as target column specifiers, they must refer to the first query's result schema.

The *result-column-lists* in the **reformat** and **normalise** clauses must specify columns of type **dateinterval** or **date**. If column names are used as target column specifiers, they must refer to the corresponding query's result schema (i.e. the specifiers in the first query's **normalise** and **reformat** clauses must refer to the first query's result schema, while specifiers in the second query's **normalise** and **reformat** clauses must refer to the second query's result schema). The **reformat** clause is executed prior to the **normalise** clause. These clauses are not allowed within subqueries.

If column names are used as target column specifiers in the **order by** clause, they must refer to the first query's result schema.

Comments

- If a column exists in more than one tables in *table-list*, then it must be qualified, if it appears in the *target-column-list*.
- If a non-qualified column name is specified in the **group by**, **reformat**, **normalise** or **order by** clauses, and such a column occurs in more than one of the tables listed in *table-list*, then it specifies the column in the leftmost table in *table-list*, containing that column.
- If a table is aliased, then the alias should be used in qualified column references that designate columns in that table, rather than the original table name.
- If a result column is renamed using either the ***`newName = select item`*** or the ***`select item as newName`*** notation, then *newName* must be used in the **group by**, **reformat**, **normalise** and **order by** clauses, in order to refer to that column. In the **reformat**, **normalise** and **order by** clauses, a target column may be specified by an integer too, as described above.
- If the **group by** clause is present, the *target-column-list* may contain only columns specified in the **group by** clause and aggregate functions. Aggregate functions include **countap**.
- If the target column list contains column references, non-aggregate functions and aggregate functions, then the **group by** clause is mandatory, and must include all target columns that do not contain aggregate functions.
- The **union all** operator followed by a *result-column-list* is semantically equivalent to the union operator, followed by the same *result-column-list*.

Examples

1. The following statement retrieves all rows from table patient.

```
select *
```

```
from patient
```

2. The following statement selects the patient codes for male patients who stayed in the hospital for more than 30 days.

```
select medrec_no
from patient
where sex = 'M'
and dur(hosp_stay_int) > 10
```

3. The following statement selects all distinct dates for which the patient with code 'ABCDEF' suffered from complication 'COMP'

```
select time
from complication
where medrec_no = 'ABCDEF'
and complication = 'COMP'
reformat as
unfold time
```

4. The following statement retrieves all drugs and the corresponding date intervals for which they were given to any patient.

```
select drug, time
from drug
reformat as
fold time
```

5. The following statement selects for every patient, the dateintervals for which the patient was hospitalised but did not suffer from any complication.

```
select medrec_no, hosp_stay_int
from patient
except 2
select medrec_no, time
from complication
```

6. The following statement retrieves the date intervals during which the patient with code 'ABCDEF' suffered from some complication but not from a non-infectious one.

```
select time
from complication
where medrec_no = 'ABCDEF'
except time
select time
from complication
where medrec_no = 'ABCDEF'
and complication in (select name from comp_list
where category = 'NON-INFECTIOUS')
```

7. The following statement displays the number of weeks that have passed since the last operation on each patient.

```
select name, windowno(max(op_date), 7, now)) as weeks
from recipient, operation
where recipient.rno = operation.rno
group by name
```

8. The following statement retrieves the total number of days that each patient was in the hospital.

```
select name, countap(stay) as total_days
from recipient
group by name
```

8. The following statement selects the patients who has had an operation during the third week after their first admission.

```
select distinct name
from recipient, operation
where recipient.rno = operation.rno
and window(min(start(stay)), 7, 3) cp interv(operation.op_date)
group by name
```

UPDATE

Syntax

update *tableName*
[portion *columnName = period*]
set *updateColumnName = newValue* {, *updateColumnName = newValue*}
[where condition]

Description

The **update** statement replaces the values of the specified columns by the values of the specified expressions for all the rows' portions that satisfy both the *condition* in the **where** clause and match with the *period* found in the **portion** clause. If either of the two clauses is omitted the updated rows or rows' portions are those specified by the existing clause. If both the **portion** and **where** clauses are omitted all the rows in the table will be replaced.

The *columnName* stated in the **portion** clause must refer to the column that the table is normalised on, and *period* must be an expression yielding a result of type **interval**. If table *tableName* is not normalised on some column, the **portion** clause may not be specified. The **portion** clause restricts the scope of the UPDATE statement to the time points included in result of *period*.

The **set** keyword is followed by a comma-separated list of *updateColumnName* = *expression* specifications, where *updateColumnName* is the name of a column occurring in *tableName*'s schema, and *newValue* is an expression yielding a result type-compatible with the *updateColumnName*'s type.

The -optional- **where** clause can be used to select the rows on which the **update** statement will operate. The *condition* specified in this clause may be any valid condition, as described in the paragraphs referring to the **select** statement.

If a primary key is defined for the table *tableName*, and the **update** statement results to having rows with matching data in the key columns, the **update** statement will fail, leaving table *tableName* unchanged. "Matching data" means identical values for key type **interval** and overlapping values for key type **point**. The **update** statement will also fail, if an attempt is made to change values to *null* in columns having the **not null** property.

Comments

- If table *tableName* is normalised on some column, and the **portion** clause is specified, then the execution of an **update** statement may alter the number of rows in the table. The number of rows may increase, decrease or remain the same.
- The *newValue* expressions may not contain qualified column references. All column references must be non-qualified and designate columns in table *tableName*.

Examples

1. Replace the time-interval '[1990-02-02, 1990-10-10)' of country A by the correct one, '[1990-01-01, 1991-01-01)'

```
update inflation
set time = '[1990-01-01, 1991-01-01)'
where country = 'A'
and time = '[1990-02-02, 1990-10-10)'
```

2. Update all employees' salary during interval '[1990-02-01, 1990-06-01)' to 110000

```
update salary
portion time = '[1990-02-01, 1990-06-01)'
set salary = 110000
```

3. Update all employees' salary during interval '[1990-02-01, 1990-06-01)' from 100000 to 110000

```
update salary
portion time = '[1990-02-01, 1990-06-01)'
set salary = 110000
where salary = 100000
```

3.6 EXTENDED SYNTAX

Commands starting with an exclamation mark (`!`) are not parsed by the VT-SQL query processor, but are passed without any modifications to INGRES, thus these commands must obey the INGRES syntax and semantic rules. This means that valid time extensions to SQL are not available to Extended Syntax statements, and if such extensions are used, the INGRES DBMS will issue error messages. The SELECT, INSERT, UPDATE, DELETE, DROP and HELP statements are not permitted as Extended Syntax statements. The end of an Extended Syntax statement is marked by a semicolon character (which is not part of a string literal or contained in a comment), or by the end of the current batch. The output of an extended syntax command is directed to the screen (in fact the standard output stream), and error messages are directed to the standard error stream, which defaults to the screen.

The contents of normalised tables must be modified only by the INSERT, DELETE and UPDATE commands. The user must not use any other command that modify the contents of tables which are normalised, as this may produce erroneous results. For example, if table n1 is created via the command

```
create table n1 (name char(20), period dateinterval
normalised (period))
```

then the command

```
! copy n1 () from 'filename'
```

should be substituted by the following commands:

```
create table ntmp (name char(20), period dateinterval)
! copy ntmp () from 'filename'
insert into n1 select * from ntmp
drop table ntmp
```

3.7 CONTROL SEQUENCES

- The control sequence `\g` can be typed at the beginning of an input line, marking the end of a batch, initiating the syntactical analysis of the input typed up to the previous line and the execution of the appropriate commands. Results are printed on the screen (or to the file to which the standard output is redirected) and error messages are directed to the standard error, which defaults to the screen.

  The commands are analysed and executed sequentially. If some command contains a syntax error, an appropriate error message is issued. Depending on the mode of the VT-SQL query processor (batch or interactive), one of the following actions will be taken when a syntax error is encountered:

  - If the VT-SQL query processor operates in interactive mode, the rest of the batch is ignored.
  - If the VT-SQL query processor operates in batch mode, execution of the program will terminate.

  After the commands are read and executed (or execution is resumed after a syntax error in interactive mode), the input buffer is cleared and the user prompted for the next batch. In batch mode, after the execution of one batch, the next batch is read from the input file.

  If and ENDOFFILE character is received before a `\g` control sequence (because the user typed `CTRL-D` or the end of the input file is reached), input read between the previous `\g` control sequence and the ENDOFFILE character is ignored. The remainder of the input lines beginning with `\g` is ignored.

- The control sequence `\r` can be typed at the beginning of an input line. This resets the input buffer, i.e. discards everything typed up to the previous line. The remainder of lines beginning with `\r` is ignored. The control sequence is not recognised unless placed at the beginning of the input line. Usage of the `\r` control sequence in batch files is permitted, but pointless.

- The control sequence `\q` can be typed at the beginning of an input line, causing the termination of the VT-SQL query processor execution. Input typed between the previous `\g` control sequence and the `\q` control sequence is ignored. The control sequence is not recognised unless placed at the beginning of the input line.

- The control sequence `\e` can be typed at the beginning of an input line. This invokes an editor, allowing modifications to the input typed up to the previous line. The editor that will be invoked is specified by the `EDITOR` environment variable, and defaults to `vi`, if the `EDITOR` variable is not set. When the user exits from the editor, the modified input is read by the VT-SQL query processor, replacing previous input. The user can, however, leave the editor without saving the changes made, leaving the VT-SQL buffer unchanged. Control sequences `\g`, `\e`, `\q` and `\r` are not recognised in the modified input, and the size of the modified input may not exceed the 10 KBytes limit. The editor can not be called when the VT-SQL query processor operates in batch mode, or when the input is redirected to a file. The control sequence is not recognised unless placed at the beginning of the input line.

  If the VT-SQL query processor is unable to invoke the editor, input typed up to the `\e` control sequence is preserved.

All control sequences are case insensitive, e.g. `\q` is equivalent to `\Q`.

3.8 RESERVED WORDS

The following words are reserved and may not be used as table names, column names, or in any other way different than the one described in section 2.2. Usage of these words within Extended Syntax statements is subject to ingres syntax and semantic rules.

```
adjacent
after
all
and
any
as
asc
avg
before
by
c
char
count
countap
covered
covers
cp
create
date
delete
desc
dist
distinct
drop
except
exists
float
float4
float8
fold
folequals
follows
from
group
having
help
in
index
insert
integer
integer1
integer2
integer4
interv
interval
into
is
key
lcovered
lcovers
like
loverlaps
max
maxdate
meets
merges
met
middle
min
mindate
normalise
normalised
not
now
null
on
or
order
overlaps
point
portion
preceds
prequals
primary
psubinterv
psupinterv
rcovered
rcovers
real
reformat
roverlaps
select
set
span
start
stop
subinterv
succ
sum
supinterv
table
tointerv
topoint
unfold
union
unique
update
values
varchar
where
window
windowno
```

3.9 ENVIRONMENT

EDITOR	This variable specifies the editor which will be called when the user types `\e`. The editor may be specified by providing the full file specification (e.g. `/usr/bin/X11/xedit`) or just the program name (e.g. `xedit`), provided that the directory in which the editor resides is included in the *path* variable. If the EDITOR environment variable is not set, it defaults to `vi`.
USER	This variable allows the VT-SQL query processor to identify the user name, which is subsequently used within queries submitted to INGRES. The USER variable is normally set and maintained by the operating system, but illegal modifications to it may result in erroneous query results.

3.10 LIMITATIONS

- The size of a query batch is limited to 10 KBytes (10240 bytes). Larger batches must be split to smaller ones, by inserting appropriate end-of-batch control sequences (`\g`). The limitation applies all kinds of input, including batch query files and input provided via editor invocation. If the size of a batch exceeds the 10 KBytes limit, execution of the VT-SQL query processor is terminated.

- Pressing CTRL-C (or otherwise generating the SIGINT signal), causes the termination of the VT-SQL query processor execution, not just the execution of the current query.

- User-level transactions are not supported.

- The VT-SQL query processor is subject to all limitations imposed by INGRES, such as maximum number of columns per table, maximum number of concurrent users, etc.

4. IMPLEMENTATION

4.1 ARCHITECTURE OVERVIEW

The ORES VT-DBMS consists of modules. Each module implements a specific task, and their operation is coordinated, in order to provide the required functionality. The overall architecture is illustrated in figure 1. Solid lines indicate transfer of control and/or data, whereas dotted lines indicate selective execution of a target module.

The functionality of the modules is described in the following paragraphs.

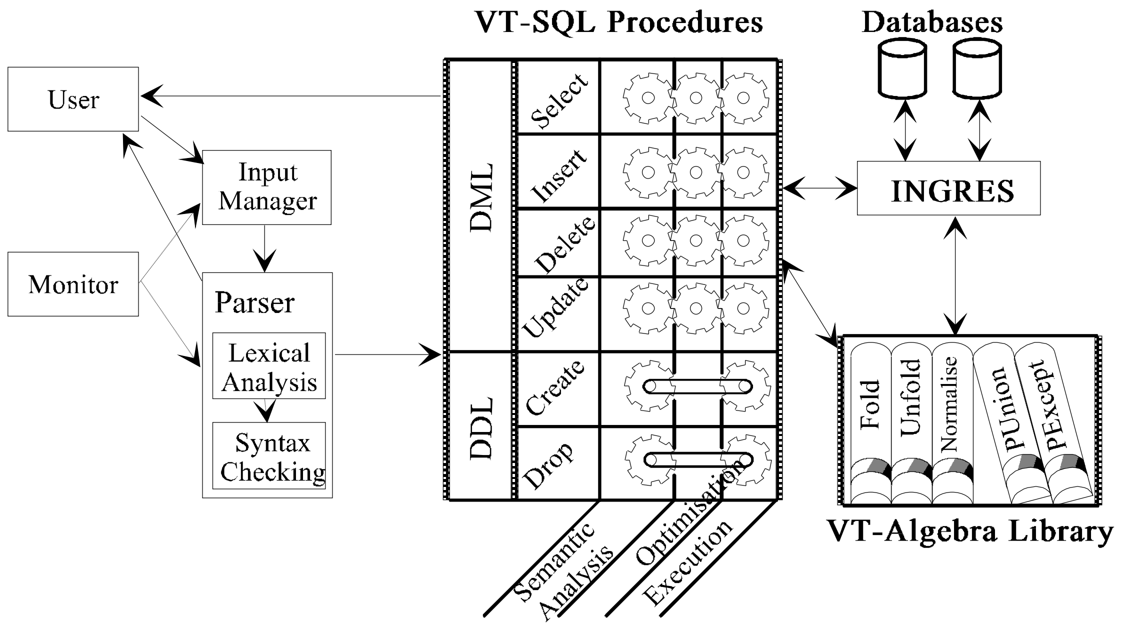


Figure 1. Overall architecture.

A. The Monitor

The monitor is responsible for initialising the VT-DBMS environment, invoking the input manager and deciding on the appropriate action, based on user input. If the user has entered a query and requested for its execution, the parser is invoked, so that the appropriate actions will be taken, and the input manager is invoked again to obtain the next query batch. If the user has requested the termination of the execution of the VT-DBMS, the monitor arranges for final housekeeping actions, releasing memory, shutting down the connection with the INGRES DBMS, etc.

B. The Input Manager

The input manager is responsible for reading the statements that will be analysed and executed. Input may be read either interactively from the user, or from a file in batch mode. Input is read in a line-oriented fashion, but in interactive mode the user may invoke a full screen editor for writing the query. The input manager accepts appropriate control sequences which mark the end of an input batch, and request either execution of the queries or termination of the execution of the VT-DBMS and sets a fla, indicating the action that must be taken. The flag is examined by the monitor.

C. The Parser

The VT-SQL parser is responsible for the syntactical analysis of the of user queries, formulation of data structures describing the queries and invocation of the appropriate execution modules, which will take the appropriate actions. Syntactical analysis is performed by two modules, the *lexical analyser* and the *syntax checker*. Data structure formulation is integrated with syntax checking for optimisation purposes.

D. The VT-SQL procedures

The VT-DBMS implementation includes one execution module for each type of VT-SQL statement. Once a statement is syntactically analysed by the parser, the appropriate execution module is called. The called module is provided with adequate parameters for statement execution and may, in turn, invoke a VT-Algebra library procedure or interact directly with INGRES. All execution modules return a status code, indicating if the operation was successful or not. In the case of failure, an error message is also returned.

E. The VT-Algebra library

The VT-Algebra library contains procedures implementing the PUNION, PEXCEPT, FOLD, UNFOLD and NORMALISE operations, which are documented in deliverables C2, C3 and C4.

F. The ORES dictionary.

The VT-DBMS requires meta-information about the tables, which is not stored in the dictionaries maintained by INGRES. This information describes the normalised columns and the primary keys defined for the tables, and is stored in a user-level dictionary consisting of two tables, namely *vtsql_norms* and *vtsql_keys*.

Table *vtsql_norms* stores information about the normalised columns in the user tables. The columns of the table are:

Column Name	Type	nulls	Description
table_name	varchar(24)	no	The name of the normalised table
user_name	varchar(9)	yes	The user name of the normalised table
column_name	varchar(24)	no	The name of the normalised table
column_sequence	integer	no	The order of normalisation. Currently, only one column may be normalised, so the value of this column is always 0.

Table *vtsql_keys* stores information about the primary key columns in the user tables. The columns of the table are:

Column Name	Type	nulls	Description
table_name	varchar(24)	no	The name of the table with a primary key
user_name	varchar(9)	yes	The user name of the table with a primary key
column_name	varchar(24)	no	The name of the primary key column
key_type	varchar(8)	no	Indicates if the field participates in the primary key with interval or point semantics
sequence	integer	no	specifies the order of the field within the primary key
index_name	varchar(24)	no	Stores the name of an index, created at INGRES level. It is used to enforce uniqueness for fields with interval semantics

4.2 DESIGN MODIFICATIONS

A. The ORES monitor

Part of the functionality assigned to the monitor in [01 P 93e] has been moved into the execution procedures. The implemented monitor interacts only with the input manager and the parser, and the parser is responsible for calling the appropriate execution module.

B. The ORES type checker

The VT-DBMS implementation does not include a separate module for type checking. Type checking in the VT-DBMS level is kept minimal, to avoid duplication of checks. (Type checking will be performed in all cases by the INGRES kernel.) The VT-DBMS performs type checks only for columns having valid time information, and these checks are built in the various execution modules.

C. The ORES optimiser

The VT-DBMS implementation does not include a separate module for optimisation. No global optimisation scheme is applicable on the VT-DBMS level, in the sense of the optimisation employed by INGRES, since neither statistical information is kept about table sizes nor indexes are defined at this level. The only possible optimisation on the VT-DBMS level is the elimination of unnecessary reformatting and normalisation operations, which is built in the relevant execution modules.

D. The ORES scheduler

The VT-DBMS implementation does not include a separate module for scheduling. Scheduling of pure SQL statements (or pure SQL parts of VT-SQL statements) is left to the INGRES kernel, whereas scheduling of various operations needed for the execution of VT-SQL statements is built in the various execution modules.

E. The ORES dictionary

The VT-DBMS implementation uses a different ORES dictionary than the one described in deliverable D3. The implementation dictionary contains a column specifying the table owner name, thus allowing for multi-user extensions, and is conferment to 3rd normal form specifications.

4.3 THE VT - SQL PARSER

The VT-SQL parser is responsible for:

(a) the syntactical analysis of the user queries

(b) determining the query type (e.g. CREATE TABLE, SELECT, etc.) and

(c) creating the appropriate data structures describing the query.

These data structures will be passed to the execution modules, in order to execute the user query. The VT-SQL parser functions are described in the following paragraphs.

Syntactical analysis

The VT-SQL syntactical analyser checks for syntactical correctness of user queries and performs various transformations, mapping user input to INGRES acceptable forms.

A. Syntax checking

Syntax checking is split into two levels. The lower level is the *lexical analyser*, which converts the stream of input characters into a stream of *tokens*, checking for illegal characters. A token may be a reserved word (e.g. **SELECT**), a string literal (e.g. **'John'**), an identifier (e.g. **employee**), a comma, or any other component of the VT-SQL language. The token stream is fed to the upper level, the *syntax checker*, which checks if the sequence in which the tokens appear conforms to the specification of the VT-SQL language. If user input is erroneous, i.e. contains illegal characters or does not form a valid VT-SQL statement, an error message is displayed.

The lexical analysis has been implemented using the *lex* lexical analyser generator tool, but a substantial part of token recognition has been coded in C, to avoid large state transition tables. The syntax checking has been implemented using the *yacc* syntax analyser generator tool.

B. Input transformations

The VT-SQL syntactical analyser performs a number of transformations on user input. These transformations are:

i) mapping of VT-SQL interval predicates to INGRES kernel function calls. Conditions of the form

```
expr1 interval_predicate expr2
```

are transformed to

```
(interval_predicate(expr1, expr2) = 1)
```

which is acceptable by the extended INGRES kernel. Conditions of the form

```
expr1 not interval_predicate expr2
```

are also accepted and transformed to

```
(interval_predicate(expr1, expr2) = 0)
```

ii) mapping of VT-SQL interval predicates involving subqueries to INGRES acceptable forms. A subquery of the form

```
expr interval_predicate subquery
```

is transformed using the following algorithm:

- if *interval_predicate* is one of *equals*, *prequals*, *preceds*, *follows* and *folequals*, it is substituted by the =, <=, <, > or >= relational operator, respectively.
- in all other cases, an appropriate error message is issued.

The inability to support more interval predicates is due to the fact that subqueries in INGRES may appear only on the right hand side of a relational operator, and may not be used as function arguments.

Subqueries of the form

```
expr not interval_predicate subquery
```

are also supported, and translated to

```
not (expr relational_operator subquery)
```

using the interval predicate substitution algorithm described above.
Both subquery forms may contain the *any* or *all* quantifiers.

iii) insertion of type coercions for functions taking arguments or returning results of type DATE. INGRES does not allow user-defined functions to accept arguments or return values of type DATE, so the kernel definitions for these functions specify string type arguments and results. The VT-SQL syntactical analyser arranges for the appropriate type casts to be performed: all arguments of type DATE are coerced to type CHAR before they are passed to the functions, using the *C* function, and all results of type DATE are coerced to that type, before they are used, using the *DATE* function. E.g. the

```
succ(date1, 5)
```

function call is transformed to

```
date(succ(c(date1), 5))
```

which converts the *date1* argument to type CHAR before passing it to the *succ* function, and coerces the function result to type DATE.

iv) argument reformatting for functions with three arguments. INGRES does not allow for functions with more than two arguments, which means that the *WINDOW* and *WINDOWNO* functions cannot be supported directly by the INGRES kernel. To overcome this problem, the kernel definitions for the WINDOW and WINDOWNO functions specify that these functions take only one argument of type CHAR, which is actually a comma-separated list of the argument values. The VT-SQL syntactical analyser arranges for passing the arguments in the appropriate format to these functions, by transforming the

```
window_function(arg1, arg2, arg3)
```

function call to

```
window function(c(arg1)+','+c(arg2)+','+c(arg3))
```

(where window_function is either *window* or *windowno*.)

v) translation of the COUNTAP aggregate function. The

```
countap(argument)
```

function call is translated to

```
sum(dur(argument))
```

Data structure formulation

In the subsequent paragraphs the following notation will be used:

- normal writing indicates reserved words.
- italic characters indicate user-provided values.
- terms enclosed in brackets (**[]**) are optional.
- terms enclosed in braces (**{}**) may occur zero or more times.
- the notation `term1 | term2` is read as "either `term1` or `term2`".
- parentheses are used to group terms. Literal parentheses are enclosed in single quotes.

A. The CREATE TABLE statement

The syntax of the CREATE TABLE statement is:

CREATE TABLE *table_name*
'('*column_name type* [NOT NULL] {, *column_name type* [NOT NULL]}
[NORMALISED *column_name*]
[PRIMARY KEY '('*column_name* [POINT | INTERVAL]
{, *column_name* [POINT | INTERVAL]}')']')'

The data structure used to describe a CREATE TABLE statement consists of the following fields:

i) table_name, which contains the table name.

ii) table_columns, which is a list of (*column_name*, *type*, *null_spec*) triplets, one for every column. *null_spec* may be either NOT NULL or empty.

iii) normalised_column, which contains the name of the column specified in the NORMALISED clause, if any, or a null value.

iv) primary_keys, which is a list of (*column_name*, *key_type*) pairs, one for every column listed in the PRIMARY KEY clause. key_type may contain one of the values POINT and INTERVAL or a null value.

Note:

The syntax of the CREATE TABLE statement has been slightly modified, compared to the syntax defined in [01P 93e]. The new syntax specifies that the -optional- NORMALISED and PRIMARY KEY clauses do not start with a comma.

B. The CREATE INDEX statement

The syntax for the CREATE INDEX statement is

CREATE [UNIQUE] INDEX *index_name* ON *table_name* (*column* {, *column*})

The data structure describing the CREATE INDEX statement consists of the following fields:

i) index_name, which contains the name of the index to be created.

ii) table_name, which contains the name of the table on which the index will be created.

iii) unique_spec, which contains an indication on whether the index will be unique or not.

iv) index_columns, which is a list containing the column names that will participate in the index.

C. The DROP TABLE statement

The syntax of the DROP TABLE statement is

DROP TABLE *table_name*

The DROP TABLE statement is described by a single field, which contains the name of the table to be dropped.

D. The DROP INDEX statement

The syntax of the DROP TABLE statement is

DROP INDEX *index_name*

The DROP INDEX statement is described by a single field, which contains the name of the index to be dropped.

E. The SELECT statement

The syntax of the SELECT statement is

extended_select
[(UNION | UNION ALL | EXCEPT) [column {, column}]
extended_select]
[ORDER BY column [ASC | DESC] {, column [ASC | DESC]]

where *extended_select* is defined as

SELECT [ALL | DISTINCT] *target_list*
FROM *table_list*
[WHERE *condition*]
[GROUP BY *column_list* [HAVING *condition*]]
[REFORMAT AS
(FOLD | UNFOLD | UNFOLD ALL) *column* {, *column*}
{(FOLD | UNFOLD | UNFOLD ALL) *column* {, *column*}}]
[NORMALISE ON *column* {, *column*}]

The SELECT statement is described by a data structure containing the following fields:

i) select1, which is a sub-structure describing the first extended select.

ii) join_type, which indicates the type of the operator joining the two extended selects (*union, union all* or *except*), if such an operator is present. A special null value indicates that only the first *extended_select* is defined.

iii) join_columns, which is a list of columns following the joining operator, if any. The list may be empty.

iv) select2, which is a sub-structure describing the second extended select. Its contents are invalid if the join_type field indicates that only the first extended select is defined.

v) order_columns, which is a list of pairs (*column_spec*, *sort_order*), describing the ORDER BY clause, if such a clause is present. If the ORDER BY clause is not defined, the field contains a null value.

The sub-structures describing the extended selects consist of the following fields:

i) quantifier, which indicates if the keywords ALL or DISTINCT are used.

ii) target_columns, which contains the list of the target columns. Each element of the list is a pair (*column_name*, *definition*), where *definition* is the expression that will be evaluated to produce the results for that column, whereas *column_name* is a name assigned to that result column for presentation reasons, using either the

```
column_name = definition
```

or the

```
definition as column_name
```

notations. If no name is defined, the column_name field contains a null value.

iii) target_relations, which is a list containing one (*relation_name*, *relation_alias*) pair for each relation specification in the FROM clause. relation_alias is optional. The syntax is

```
relation_name alias_name
```

iv) where_clause, which contains the condition following the WHERE keyword.

v) group_spec, which contains the GROUP BY and HAVING clauses.

vi) reformat_spec, which is a list describing the REFORMAT clause. Each element in the list consists of an indication of the reformat operation specified (FOLD, UNFOLD, UNFOLD ALL) and a list of the columns designated for that operation.

vii) normalise_columns, which is a list of the column names specified in the NORMALISE clause.

F. The INSERT statement

The syntax of the INSERT statement is

INSERT INTO *table_name* ['(' *column* {, *column*} ')']
insert_values_spec

where insert_target may be either

VALUES (value {, value})

or an extended select (see paragraph E).

The INSERT statement is described by a data structure containing the following fields:

i) table_name, which contains the name of the target table.

ii) insert_idents, which is a list containing one element for each of the parenthesised identifiers immediately following the table name.

iv) insert_values, which is a list of the values defined in the VALUES clause.

v) insert_subquery, which is a data structure describing the extended select statement.

G. The UPDATE statement

The syntax of the UPDATE statement is

```
UPDATE table_name
[PORTION column = expr]
SET column = expr {, column = expr}

[WHERE condition]
```

The data structure used to describe the UPDATE statement consists of the following fields:

i) table_name, which contains the name of the table to be updated.

ii) portion_spec, which contains a pair (*column*, *expr*) describing the PORTION clause, if such a clause is present.

iii) assignments, which is a list of (*column*, *expr*) pairs, one for each assignment following the SET keyword.

iv) condition, which contains the condition in the WHERE clause.

H. The DELETE statement

The syntax of the DELETE statement is

```
DELETE FROM table_name
[PORTION column = expr]
[WHERE condition]
```

The data structure used to describe the DELETE statement consists of the following fields:

i) table_name, which contains the name of the target table.

ii) portion_spec, which contains a pair (*column*, *expr*) describing the PORTION clause.

iii) condition, which contains the condition in the WHERE clause, if the clause is present, or the null value.

I. The HELP statement

The syntax of the HELP statement is

HELP [object]

The HELP statement is described by a single field, containing the name of the object or a null value.

J. The escaped DBMS statements

If a statement does not contain any of the characteristics of VT-SQL then its execution time may be minimised in the statement is routed directly to INGRES. This can be achieved if the statement is preceded by a "!", i.e.

! *command*

Note : A "!" should not precede a SELECT, INSERT, DELETE, UPDATE, DROP or HELP statement.

Implementation notes

A. Memory allocation monitoring

For optimisation purposes, syntactical analysis and data structure formulation have been integrated, thus allowing for user input to be scanned only once. Data structure fields are filled in as soon as the appropriate tokens are recognised by the syntactical analyser. In the case, however, that a syntactical error is found, the parsing procedure is terminated abruptly and care is taken so that memory allocated for data structure field storage is freed. In order to tackle this problem, memory allocation during parsing is monitored, and the monitor is called to free the allocated memory, in the case that a syntactical error occurs.

B. Escaped commands

The underlying DBMS is responsible for lexical and syntactical analysis of the escaped commands. No tokenisation, illegal character checking or syntax checking is performed for escaped commands.

4.2 VT-SQL DDL STATEMENTS IMPLEMENTATION

CREATE TABLE statement

According to SQL2, whenever a table is created, its name, attributes and primary key (if any) must be specified. In order to support the distinction between valid time intervals and time intervals in VT-SQL, we extended the syntax of the *create table* statement as follows:

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

All column names specified in <normalised-column-list> are of type DATEINTERVAL. When data are inserted, deleted or updated, the table is normalised according to these specified intervals. Normalisation occurs in the order the columns appear after the NORMALISED keyword. All valid time intervals should participate in the PRIMARY KEY section (if a primary key is defined) followed by the keyword POINT. Other columns may also be present in this section when followed by the keyword INTERVAL which (which is a default, if not declared).

The supporting of the extended functionality may demand the construction of specific structures (e.g. indexes) as well as the provision of additional system tables. Both the normalisation schema of every table created through VT-SQL and the columns participating in the primary key are maintained in the new system tables implemented, namely *vtsql_norms* and *vtsql_keys*.

Due to the complexity originating from the extended functionality, the execution phase demands an exhaustive checking in order to ensure the given CREATE TABLE statement complies with the provided specifications. The extended system tables are to be updated during the execution phase. The overall actions performed are presented in the following algorithm.

A. Normalisation attributes are checked for the cases of not being present in the table column list, not being of type DATEINTERVAL, being duplicated in the normalisation column list and, in combination with the primary key declaration section, not being present as primary keys or not being declared of type POINT. The above checking results in the appropriate error messages whenever any of the conditions described is not fulfilled.

B. Primary key attributes are examined for the cases of not being present in the table column list, being duplicated in the primary key column list and, in combination with the normalisation columns declaration section, not being present as normalisation attributes.

C. The appropriate SQL CREATE TABLE statement is formed and executed. Declaration of table columns participating in the PRIMARY KEY or in NORMALISED section is modified when columns are declared as nullable, so that they always acquire the *not null* property.

D. The appropriate SQL CREATE INDEX statements are formed and executed in case that existence of unique indexes is required.

E. Normalisation attributes are stored in the appropriate system table, namely *vtsql_norms.*

F. Primary keys are stored in the corresponding system table, namely *vtsql_keys*.

DROP TABLE statement

The syntax of the VT-SQL DROP TABLE statement remains the same as the analogous SQL statement; thus, the syntax becomes:

DROP TABLE *table-name*

The functionality of the DROP statement corresponds to the extended features embedded in the CREATE TABLE operation.

During the DROP TABLE statement execution, all entries referring to table *table-name* are removed from the systems tables *vtsql_norms* and *vtsql_keys*, if they have been recorded in them. The appropriate SQL queries are issued for this purpose. Accordingly, the table itself is dropped through the SQL DROP statement. We note that INGRES itself also takes care of all structures, as indexes, relating to this table that is, all indexes previously declared are removed.

CREATE INDEX statement

As already stated, the syntax of the CREATE INDEX statement is

CREATE [UNIQUE] INDEX *index_name*
ON *table_name* (*column* {, *column*})

Once the parser has recognised a CREATE INDEX statement, the appropriate execution module is invoked. The execution module is provided with parameters that are sufficient for the reconstruction of the original statement, which is forwarded to INGRES for execution. Errors that may occur during statement execution (such as the specification of a non-existent column or an attempt to create a UNIQUE index on a table that contains duplicates in the specified columns) are trapped, and appropriate error messages are displayed.

DROP INDEX statement

As already stated, the syntax of the DROP INDEX statement is

DROP INDEX *index_name*

Once the parser has recognised a DROP INDEX statement, the appropriate execution module is invoked. The execution module is provided with a single parameter, namely the name of the index to be dropped. Errors that may occur during statement execution (such as attempting to drop a non-existent index) are trapped, and appropriate error messages are displayed.

4.5 VT-SQL DML STATEMENTS IMPLEMENTATION

SELECT statement

The syntax of the SELECT statement is as follows:

SELECT [DISTINCT | ALL] *target-column-list*
FROM *table-list*
[WHERE *condition*]
[GROUP BY *column-list*]
[HAVING *condition*]]
[REFORMAT AS {(FOLD | UNFOLD | UNFOLD ALL) *result-column-list*}]
[NORMALISE ON *result-column-list*]
[(UNION | UNION ALL | EXCEPT) [*result-column-list*]
SELECT [DISTINCT | ALL] *target-column-list*
FROM *table-list*
[WHERE condition]

[GROUP BY *column-list*]
[HAVING *condition*]]
[REFORMAT AS {(FOLD | UNFOLD | UNFOLD ALL) *result-column-list*}]
[NORMALISE ON *result-column-list*]]
[ORDER BY *result-column-list*]

Thus, a SELECT statement may be a select query, extended by the *reformat* and *normalise* clauses, or two extended select queries, joined with a *union*, *union all* or *except* operation. The joining operations may include valid time semantics, which is indicated by a result column list immediately following the operator.

Once the parser has recognised a SELECT statement, the appropriate execution module is invoked, in order to evaluate the query and present the results to the user. The execution module is provided with parameters allowing semantic analysis, type checking, optimisation and query evaluation. All these steps are described in the following paragraphs.

Semantic analysis

The semantic analyser built in the select execution module examines the query definition, as presented by the parser, and determines the actions that must be taken in order to evaluate the query. The following algorithm is used to determine the appropriate actions:

A. If the user query consists of one *simple query* or two simple queries joined via a *union* or *union all* operation with no valid time semantics, then the query can be forwarded directly to INGRES for execution. (A *simple query* is a select statement consisting only of an attribute list, the FROM clause and, possibly, the WHERE and GROUP BY/HAVING clauses.) In this case, the optimisation step is skipped.

B. If the user query consists of one query which includes a REFORMAT or NORMALISE clause, then the following procedure is introduced:

   i) the simple query is extracted. The types of the columns appearing in the REFORMAT and NORMALISE clauses are checked, and if an illegal operation is detected (i.e. an operation on a column whose type is neither DATE nor DATEINTERVAL), query evaluation is aborted. If no illegal operation is detected, the simple query is forwarded to INGRES for evaluation and execution. The results are stored in a temporary table.

ii) if the user query includes a REFORMAT clause, then the optimisation module is called, in order to eliminate redundant operations. At execution time, the **fold** and **unfold** valid time algebra library procedures are invoked, depending on the operation stated in the REFORMAT clause. Each invocation operates on the intermediate table produced by the previous one (the first invocation operates on the temporary table produced by step i), producing a new intermediate table. Intermediate tables are dropped as soon as the next invocation is completed (e.g. the intermediate table created by the first invocation is dropped as soon as the second invocation is completed).

iii) if the user query includes a NORMALISE clause, then the optimisation module is called to eliminate redundant operations. At execution time, the **normalise** valid time algebra library procedure is invoked, operating on the last intermediate table produced by step ii (or the intermediate table produced by step i, if the user query does not include a REFORMAT clause). The previous temporary table is then dropped.

Finally, the results are retrieved from the last intermediate table that was produced in steps (i) to (iii) and presented to the user.

C. If the user query consists of two select statements, joined by a *union*, *union all* or *except* operation, then the following procedure is used:

i) the schema of the result table of each query is determined, and the two schemata are checked for compatibility. If the two schemata are incompatible, query evaluation is aborted. If the operator joining the two queries is followed by a result column list, the types of the columns appearing in the list are checked. If the type of any of the columns is neither DATE nor DATEINTERVAL, query evaluation is aborted. If this step completes successfully, then the following steps take place:

ii) the first query is evaluated and executed, using steps (i) to (iii) of case B, producing an intermediate table. If a runtime error occurs during its execution, the whole procedure is aborted.

iii) the second query is evaluated and executed, using steps (i) to (iii) of case B, producing a second intermediate table. If an error occurs during its execution, the whole procedure is aborted.

iv) if the queries are joined by a *union* or *union all* operation not followed by column names, then a query retrieving the union of the tuples in the intermediate tables produced in steps (ii) and (iii) above is formulated and forwarded to the execution module. Finally, the intermediate tables produced by steps (ii) and (iii) are dropped.

v) if the queries are joined by an except operation, or a union or union all operation which are followed by column names, the appropriate valid time algebra library procedure is invoked, producing a third intermediate table. The valid time algebra library procedure is selected as follows:

   a) if the joining operator is an *except* with no valid time semantics, then the **except** valid time algebra library procedure is used.

   b) if the joining operator is an *except* with valid time semantics, then the **pexcept** valid time algebra library procedure is used.

   c) if the joining operator is a *union* or **union all** with valid time semantics, then the **punion** valid time algebra operator is used.

   Afterwards, the intermediate tables produced in steps (ii) and (iii) are dropped, and a query retrieving the contents of the third intermediate table is forwarded to the execution module. The third intermediate table is finally dropped.

Optimisation

The optimisation procedure is invoked to remove redundant operations defined by the REFORMAT and NORMALISE operations. The optimisation procedures used for these clauses are described in the following paragraphs.

REFORMAT clause optimisation

The REFORMAT clause optimisation procedure rearranges the reformats and removes some unnecessary reformats. Recall that a REFORMAT clause contains an arbitrary list of FOLD, UNFOLD and UNFOLD ALL operations, each one applicable on a set of result columns. Rearranging implies the following:

1) UNFOLD ALL operations immediately followed by UNFOLD operations are changed to UNFOLD operations. This does not alter the semantics of the operation, as duplicates produced by the UNFOLD ALL operation will be removed by the following UNFOLD operation.

2) UNFOLD ALL operations immediately following UNFOLD operations and followed by FOLD or UNFOLD operations are changed to UNFOLD operations. This does not alter the semantics of the operation, as duplicates produced by the UNFOLD ALL operation will be removed by the following FOLD or UNFOLD operation.

3) consecutive FOLD operations are packed in a single one.

4) consecutive UNFOLD operations are packed in a single one.

5) consecutive UNFOLD ALL operations are packed in a single one.

Steps (1) and (2) allow for better packing of operations, preserving operation semantics. Packing the operations into bigger chunks allows for exploitation of optimised code in the valid time algebra library procedures: if these procedures are coded in such a way that folding or unfolding of multiple columns is done in a single pass (i.e. with one reading of the table), then the execution procedure can benefit from the packing, since less intermediate steps will be taken. The packing of operations also eases the work of the next optimisation step, the removal of unnecessary operations, which scans every operation, and removes duplicate occurrences of the same column, if any are found. Figure 1 illustrates the two steps of the optimisation procedure.

```
reformat as           reformat as          reformat as               reformat as
     fold 1                fold 1               fold 1, 2, 1              fold 1, 2
     fold 2, 1             fold 2, 1            unfold 3, 4, 5, 4, 3      unfold 3, 4, 5
     unfold 3, 4           unfold 3, 4          fold 6                    fold 6
     unfold all 5, 4       unfold 5, 4          unfold 7, 8, 8, 9         unfold 7, 8, 9
     unfold all 3          unfold 3             fold 10                   fold 10
     fold 6                fold 6               unfold all 11             unfold all 11
     unfold all 7, 8       unfold 7, 8
     unfold all 8, 9       unfold 8, 9
     fold 10               fold 10
     unfold all 11         unfold all 11

       (a)                   (b)                    (c)                       (d)
```

Figure 1. (a) An original REFORMAT clause. (b) The same REFORMAT clause after changing UNFOLD ALL operations to UNFOLD operations. (c) The same REFORMAT clause after operation packing. (d) The same REFORMAT clause after duplicate removal.

NORMALISE clause optimisation

The optimisation of the NORMALISE clause removes duplicate references to the same column (except the first occurrence of a column name, all other are eliminated), and thus normalisation of each column is performed only once.

Query evaluation

The evaluation procedure built in the select execution module is responsible for carrying out the actions scheduled by the semantic analysis procedure (which may have been modified by the optimisation module). The following points are taken into consideration for query execution:

A. Column renaming

If the query evaluation procedure requires the creation of an intermediate table to store the results of a simple query, care must be taken so that column names for the intermediate table will be unique. The intermediate table cannot be created by simply transforming the simple query to

```
create table temp_name as simple_query
```

because if the target column list of *simple_query* contains two or more columns with the same name, the statement will be rejected by INGRES. To avoid such problems, the columns in the target list of the simple query are renamed, and given unique internal names. The original names of the columns in the target list are saved, and used when the results are presented to the user. If the target list is a star (*), the actual target columns are calculated and then renamed.

Since the columns in the intermediate tables have different names than the ones in the original tables, care must be taken so that the **fold**, **unfold** and **normalise** valid time algebra library procedures are called with the appropriate arguments, i.e. the internal attribute names. The target column list which follows the ORDER BY clause is also changed, and attribute names are transformed to column sequence numbers. Figure 2 illustrates the functionality of the column renaming scheme.

```
select s.sno, sp.sno, time = sp.period
from s, sp
where s.sno > sp.sno


reformat as unfold sp.period


order by s.sno, sp.sno
```

(a)

```
create table temp0 as
select    attr0 = s.sno, attr1 = sp.sno,
          attr2 = sp.period
from s, sp
where s.sno > sp.sno


temp1 = unfold [attr2] (temp0)


select sno = attr0, sno = attr1, time = attr2
from temp1
order by 1, 2
```

(b)

Figure 2. (a) A query whose evaluation requires the creation of an intermediate table. (b) The actual operations performed, using the renaming scheme (destruction of intermediate tables is not included).

B. Exception handling

During query evaluation certain exceptions may occur, such as passing a negative argument to the *log* function, or using the *intervsect* function with two non-overlapping dateintervals as arguments. The INGRES kernel will notify the application retrieving the data, as soon as the row containing the invalid data is examined, which means that previous rows will be presented to the application with no error indication.

It is clear that a terminal monitor, which presents query results to the user, should not issue an error message, informing the user that an exception has occurred, after displaying some rows. The expected behaviour is that either the query is correct, so its results are displayed, or an exception occurs during its evaluation, in which case only the error message is printed. This can be accomplished in two ways:

i) result rows are fetched and stored in memory data structures. If a row contains invalid data, INGRES will raise an exception, in which case memory used to store previously fetched rows is freed, and query evaluation is aborted. If all rows are successfully fetched, then the result is displayed.

ii) the result rows are fetched, but not stored into memory. Fetching the rows will cause INGRES to raise an error if a row containing invalid data is encountered, in which case query evaluation is aborted. If all rows are valid, they are fetched once more into memory, one after another, and printed.

The two approaches have advantages and disadvantages. The first approach is clearly more efficient, since rows are fetched into memory once, but requires large amounts of memory. This can cause problems with computers with small memories, especially in a multiuser environment, where many users may submit select queries simultaneously.

The second approach is slower, since rows must be fetched twice, but is more stable, with respect to query result sizes and number of active users. It is noted that the performance penalty paid for fetching the result rows twice, will not be extremely high, since the data will be present in INGRES buffers (if the results are small enough to fit entirely in those buffers), where INGRES can rapidly access them.

The implementation takes the second approach, but two optimisations limit the number of cases in which the results are actually fetched twice to a minimum:

- if the query does not contain references to function that may cause exceptions, there is no need to check the rows before displaying them. The parser sets a flag, indicating the existence of such function references. If no such function reference occurs in the user query, the rows are fetched one after another and printed. In the case, however, that such function references do occur, all rows must be examined before any of them is displayed.

- if the query evaluation procedure requires creation of intermediate tables, all possible exceptions will be raised during the creation of these tables, because all functions are evaluated at this point. Thus, if the results are retrieved from an intermediate table, there is no need to examine the rows before printing them, so all rows are fetched into memory once.

INSERT statement

The syntax of the VT-SQL insert statement remains exactly the same as in SQL2. However, the actual insertion method depends on the table normalisation schema and primary key definition, both defined within the CREATE TABLE statement. Thus, the syntax has either of the following forms:

INSERT INTO *table-name* [(*column-name-list*)]
VALUES *value-list*

or

INSERT INTO table-name [(column-name-list)] subselect
where *subselect* is defined as in the select statement.

If the table does not contain valid time intervals, INSERT operation is identical to the corresponding SQL2 operation. If a key has been declared, according to SQL2 the insertion fails when the key value in one of the rows to be inserted matches the key value of any existing row or when the key values of two or more of the insertion rows are the same. The functionality of the SQL2 INSERT statement is not only preserved within VT-SQL, but also extended. If a valid time interval is participating in the key, the two rules mentioned above must be satisfied in a time-point granularity. In case the table contains valid time intervals, it remains normalised after the insert operation. Processing and execution of the *subselect* statement conforms with the corresponding actions in select operation.

Execution of the INSERT statement is viewed as a complex task since it demands the analysis and execution of the included subselect statement. In case only insertion values are present the number of the steps required is minimised. The overall algorithm may be described in the following:

Checking for the existence of a subselect statement that is to be processed before any other actions are performed. These are the steps required for processing the subselect:

A. Examination of the subselect for the case that all columns are to be retrieved through a “select * “ clause, whereas the NORMALISED or REFORMAT clauses are not empty. In this case, further processing is aborted. (Similarly to GROUP BY, the column names after any of these clauses must be explicitly referenced at the SELECT clause).

B. Preparation of the subselect statement execution.

C. Examination of the resulting table. In case that duplicate column names are found, duplicating columns (i.e. columns having names identical to previous ones) are renamed appropriately.

D. Reformatting of the resulting table according to the attributes and the reformat type specified in the REFORMAT AS cause.

E. Normalisation of the resulting table according to the attributes included in the NORMALISED ON clause.

For these two latter actions we may note that the attributes present in both clauses are replaced with a number indicating the order of the column in the resulting table. This is performed because, under certain circumstances, referencing the resulting table columns through their names cannot always be achieved.

The actions performed when only autonomous values are to be inserted, are the following:

A. Creation of a empty table identical to the insertion table.

B. Insertion of the values specified in the previously created table.

From this point all the necessary actions are common for both cases. So far, a table is created, the all the rows to be inserted.

Examining whether a normalisation should be performed after the rows are inserted. This information is acquired by accessing the *vtsql_norms* system table. If no normalisation is necessary, rows are simply inserted, thus completing the operation.

In normalisation is required, an extensive primary key violation check is performed, as specified in [01P 93e].

Insertion of the values contained in the resulting table in the insertion table. The whole table is subsequently normalised on the attributes specified in the NORMALISED clause of the CREATE TABLE statement.

DELETE statement

The DELETE statement is as follows:

DELETE FROM *table-name*
[PORTION *key-point-column-name* = *dateinterval*]
[WHERE *condition*]

Thus a DELETE statement is a delete query, extended by the PORTION clause.

Once the parser has recognised the DELETE statement, the appropriate execution module is invoked, in order to execute the query and make the proper deletions to the table. The execution module is provided with parameters which will allow for semantic analysis, type checking, optimisation and query execution. In the following paragraphs, these steps are analysed.

The semantic analyser built in the delete execution module examines the query definition, as presented by the parser, and determines the actions that must be taken in order to execute the query. The following algorithm is used:

A. If the user query concerns a deletion from a table, not recorded in the ORES Dictionary, then the only check made is to ensure that no PORTION clause exists. If such a clause does exist, a syntax error is identified, otherwise the query is forwarded to INGRES for execution.

B. If the user query concerns a deletion from a table recorded in the ORES Dictionary, then following procedure is used:

   i) If no PORTION clause is contained in the user query, the query is passed to INGRES for execution.

   ii) If a PORTION clause is contained in the user query, it is initially checked whether the column name, after the keyword PORTION represents a column with respect to which a normalisation has to take place (such a column has previously been declared in the CREATE TABLE statement). If this is not the case, then a syntax error is identified, therefore further processing is aborted and a message is displayed. In the opposite case, a check is made that the rows or portions of rows that are about to be deleted really exist (i.e. that there do exist rows that satisfy the WHERE clause condition and have common points with the dateinterval found in the PORTION clause), If no such row exists, then the query execution is finished by displaying a "No rows deleted" message. If such rows do exist, then the remainder of the algorithm described in [01P 93e] is executed and the portions of these rows are eliminated.

UPDATE statement

The syntax of the UPDATE statement is as follows:

```
UPDATE table-name
[PORTION key-point-column-name = dateinterval]
SET updatable-column-list
[WHERE condition]
```

Similar to the DELETE statement, an UPDATE statement is an update query, extended by the PORTION clause.

Once the parser has recognised the UPDATE statement, the appropriate execution module is invoked, in order to execute the query and updates the table.

The semantic analyser built in the update execution module examines the query definition, as presented by the parser, and determines the actions that must be taken in order to execute the query.

The algorithm is similar to that for deletion, (except that now an update takes place instead of a deletion) and is in accordance with the algorithm described in [01P 93e].

HELP statement

The syntax of the HELP statement is

HELP [*object_name*]

Once the parser has recognised a HELP statement, the appropriate execution module is invoked. The execution module may be provided with a single parameter, whose value is the name of the object on which help is requested.

If no object name is specified, then a list of the tables which the user can access is retrieved and displayed. To determine the list, the system catalogue `iitables` is queried to retrieve the tables that are owned by the user, while the system catalogue `iipermits` is queried to retrieve the tables owned by other users, but are accessible by the user issuing the HELP statement.

If an object is specified, it must designate a system catalogue, a user table or an index. The system catalogue `iicolumns` is queried to determine the names, type and properties of the columns belonging to the object. Normalisation and primary key information is retrieved from the VT-DBMS system catalogues, `vtsql_norms` and `vtsql_keys`, respectively.

5. CONCLUSIONS

In this report we presented the valid time SQL (VT-SQL) which has been developed in the ORES project. The evaluation tests have shown that the software runs according to the specification. The results are presented in appendix A.

APPENDIX A: VT-SQL EVALUATION

1. Introduction

The present appendix contains the tests of VT-SQL. The evaluation has been tailored to examine the correctness of the functionality of VT-SQL. The functionality of VT-RA and the INGRES kernel has already been tested (Appendix A, deliverable C4). A sample database has been used and thorough tests have been made on all VT-SQL statements. In what follows, the description of the sample database is given, and tests made on CREATE TABLE, INSERT, DELETE, UPDATE, SELECT are presented, in that order.

2. Sample database

part of the time test has been made against the tables which follow. All these tables contain valid time data and care has been taken for them to contain duplicate rows and to be non-normalised. Similar tables containing invalid data have also been used.

(S)ALARY

Name	Amount	Days
John	10000	[1993-06-02, 1993-06-06)
John	10000	[1993-06-09, 1993-06-12)
John	12000	[1993-06-15, 1993-06-18)
Alex	14000	[1993-06-09, 1993-06-12)

(A)SSIGNMENT

Name	Dept	Days	Type
John	shoe	[1993-06-03, 1993-06-07)	salesman
John	food	[1993-06-07, 1993-06-11)	supervisor
John	toys	[1993-06-11, 1993-06-15)	salesman
Alex	shoe	[1993-06-05, 1993-06-10)	supervisor
Mary	toys	[1993-06-05, 1993-06-11)	supervisor

(D)uplicated SALARY

Name	Amount	Days
John	10000	[1993-06-02, 1993-06-06)
John	10000	[1993-06-02, 1993-06-06)
John	12000	[1993-06-15, 1993-06-18)
Alex	14000	[1993-06-09, 1993-06-12)

(N)on Normalised ASSIGNMENT

Name	Dept	Days	Type
John	shoe	[1993-06-03, 1993-06-07)	salesman
John	food	[1993-06-07, 1993-06-11)	supervisor
John	food	[1993-06-08, 1993-06-15)	salesman
Alex	shoe	[1993-06-05, 1993-06-10)	supervisor
Mary	toys	[1993-06-05, 1993-06-11)	supervisor

(I)nflation

Country	Percentage	Period
greece	0.15	[1993-06-01, 1993-06-02)
spain	0.10	[1993-06-03, 1993-06-04)
EEC	0.08	[1993-06-07, 1993-06-08)

(O)vertive

Name	Date	No of Hours
John	1993-06-02	2
John	1993-06-03	2
John	1993-06-04	2
John	1993-06-05	2
John	1993-06-06	2
John	1993-06-15	1
John	1993-06-20	2
Alex	1993-06-20	2

Dup(L)icated Overtime

Name	Date	No of Hours
John	1993-06-02	2
John	1993-06-03	2
John	1993-06-15	1
John	1993-06-15	1
Alex	1993-06-20	2

(C)omplication

Patient	Complication	Period
John	Hypotassemia	[1993-06-01, 1993-06-05)
John	Hyperglykemia	[1993-06-01, 1993-06-09)
John	Leykopenia	[1993-06-01, 1993-06-10)
John	Hypomagnesemia	[1993-06-01, 1993-06-15)
John	Dialysis	[1993-06-01, 1993-06-20)
John	Atelectasis	[1993-06-08, 1993-06-10)
John	Hemothorax	[1993-06-08, 1993-06-15)
John	Pneumothorax	[1993-06-08, 1993-06-20)
John	Ultrafiltration	[1993-06-09, 1993-06-11)
John	Psychosis	[1993-06-12, 1993-06-15)
John	Pancreatitis	[1993-06-12, 1993-06-20)
John	Bone	[1993-06-14, 1993-06-20)
John	Hemolytic Anemia	[1993-06-17, 1993-06-20)

S2

Name	Amount	Date
John	10000	[1993-06-17, 1993-06-20)

D2

Name	Amount	Date
John	10000	[1993-06-17, 1993-06-20)
John	10000	[1993-06-17, 1993-06-20)
John	12000	[1993-06-15, 1993-06-18)
Alex	14000	[1993-06-09, 1993-06-12)

(SO)

Name	Amount	Days	Hours	Date
John	10000	[1993-06-02, 1993-06-06)	2	1993-06-02
John	10000	[1993-06-09, 1993-06-12)	2	1993-06-03
John	10000	[1993-06-09, 1993-06-12)	2	1993-06-03
John	12000	[1993-06-05, 1993-06-12)	1	1993-06-15
Alex	14000	[1993-06-09, 1993-06-12)	2	1993-06-20

(OO)

Name	Date	Hours	Period
John	1993-06-02	2	1993-06-02
John	1993-06-03	2	1993-06-02
John	1993-06-04	2	1993-06-02
John	1993-06-04	2	1993-06-03
Alex	1993-06-20	2	1993-06-03

(SOD)

Name	Amount	Days	Hours	Date
John	10000	[1993-06-02, 1993-06-06)	2	1993-06-02
John	10000	[1993-06-09, 1993-06-12)	2	1993-06-03
John	10000	[1993-06-09, 1993-06-12)	2	1993-06-03
John	12000	[1993-06-05, 1993-06-12)	1	1993-06-15
Alex	14000	[1993-06-09, 1993-06-12)	2	1993-06-20

(OOD)

Name	Date	Hours	Period
John	1993-06-02	2	1993-06-02
John	1993-06-03	2	1993-06-02
John	1993-06-03	2	1993-06-02
John	1993-06-04	2	1993-06-03
Alex	1993-06-20	2	1993-06-03

(DA)

Name	Amount	Dept	Days	Period
John	10000	shoe	[1993-06-02, 1993-06-06)	[1993-06-03, 1993-06-07)
John	10000	food	[1993-06-02, 1993-06-06)	[1993-06-03, 1993-06-07)
John	10000	food	[1993-06-09, 1993-06-12)	[1993-06-07, 1993-06-11)
John	12000	toys	[1993-06-15, 1993-06-18)	[1993-06-11, 1993-06-15)
Alex	14000	shoe	[1993-06-09, 1993-06-12)	[1993-06-05, 1993-06-11)

(E)

Name	Amount	Date

(DAN)

Name	Amount	Dept	Days	Period
John	10000	shoe	[1993-06-02, 1993-06-06)	[1993-06-03, 1993-06-07)
John	10000	food	[1993-06-02, 1993-06-06)	[1993-06-07, 1993-06-11)
John	10000	food	[1993-06-02, 1993-06-06)	[1993-06-08, 1993-06-15)
John	12000	toys	[1993-06-15, 1993-06-18)	[1993-06-11, 1993-06-15)
Alex	14000	shoe	[1993-06-09, 1993-06-12)	[1993-06-05, 1993-06-11)

3. Evaluation of CREATE TABLE

Four main categories of tests have been made:

1. Tables without *primary key* or *normalised* clause. All specified data types

   - integer
   - integer1
   - varchar() or char() or c()
   - float
   - date or point and
   - dateinterval

have been tested

Test Result: According to the specifications

2. Tables with *primary key* clause. Tests have been made, aiming to evaluate the correctness of the parser for possible combinations of column names, participating in the primary key.

Test Result: According to the specifications

3. Tables with *normalised* clause. Tests have been made, aiming to evaluate the correctness of the parser for possible combinations of column names, participating in the NORMALISED clause.

Test Result: According to the specifications

3. Tables with *normalised* and *primary key* clause. Tests have been made, aiming to evaluate the correctness of the parser for possible combinations of normalised column names, which also participate in key.

Test Result: According to the specifications

As far as we can estimate, these tests have been exhaustive.

4. Evaluation of INSERT, DELETE, UPDATE

The ecaluation tests of the above statements have been separated in four major categories. Each category corresponds to each table type, namely:

1. Table without *primary key* and *normalised* clause.
2. Table with *primary key* clause.
3. Table with *normalised* clause.
4. Table with *primary key* and *normalised* clause.

4.1. Table without *primary key* and *normalised* clause

The table *salary* was created and a set of statements have been issued to perform the testing.

```
create table salary
(name char(10),
sal float,
time dateinterval not null)
\g

insert into salary (name, sal) values('John', 500000)

insert into salary values('John', 170000, '[1990-01-01, 1992-01-01)')

insert into salary values('John', 2 * 100000, '[1992-01-01, 1994-01-01)')

insert into salary values('John', 200000, tointerv(now()))

insert into salary values('John', 300000, interv(start('[1993-01-01, 1994-01-01)', now()))

insert into salary (sal, time) values (170000, '[1990-01-01, 1992-01-01)')

insert into salary (sal, time) values (2 * 100000, '[1992-01-01, 1994-01-01)')

insert into salary (sal, time) values(200000, tointerv(now()))

insert into salary (sal, time) values(300000, interv(start('[1993-01-01, 1994-01-01)',
    now()))

insert into salary values('John', 170000, '[1990-01-01, 1992-01-01)')

insert into salary (sal, time) values (2 * 100000, '[1992-01-01, 1994-01-01)')

insert into salary (select name, amount, days from s)

insert into salary(sal, time) select amount, time from s
insert into salary select name, amount, days from s
where amount = 10000 and
start(days) < stop(interv(stop(days), now()))
reformat as
unfold time
fold time

insert into salary (sal, time) select amount, days from s
where amount = 10000 and
start(days) < stop(interv(stop(days), now()))
reformat as
unfold time
fold time

delete from salary

insert into salary values('John', 170000, '[1990-01-01, 1992-01-01)')

insert into salary values('John', 2 * 100000, '[1992-01-01, 1994-01-01)')

insert into salary values('John', 200000, tointerv(now()))

insert into salary values('John', 300000, interv(start('[1993-01-01, 1994-01-01)', now()))

insert into salary (sal, time) values (170000, '[1990-01-01, 1992-01-01)')

insert into salary (sal, time) values(2 * 100000, '[1992-01-01, 1994-01-01)')

insert into salary (sal, time) values(200000, tointerv(now()))

insert into salary (sal, time) values(300000, interv(start('[1993-01-01, 1994-01-01)',
    now()))

insert into salary select name, amount, days from s

insert into salary (sal, time) select amount, days from s

delete from salary where amount = 10000

delete from salary where amount = 10000 or name = ''

delete from salary
where interv(start(time), succ(now(), span(now(), start(time))) cp
interv(stop(time), stop(merge(time, interv(start(time), now()))))
```

```
insert into salary values ('John', 170000, '[1990-01-01, 1992-01-01)')

delete from salary portion time = '[1990-01-01, 1991-01-01)

delete from salary

select * from salary

insert into salary values ('John', 100000, '[1985-01-01, 1987-01-01)')

insert into salary values ('John', 130000, '[1987-01-01, 1989-01-01)')

insert into salary values ('John', 160000, '[1989-01-01, 1990-01-01)')
update salary
set sal = 170000
where time = '[1989-01-01, 1990-01-01)'

update salary
set time = '[1990-01-01, 2000-01-01)
where interv(start(time), now()) equals
interv(start(time), stop(interv(stop(time), now())))

update salary
portion time = '[1999-01-01, 1999-02-02)'
set sal = 200000
where name = 'John'

update salary
set name = 'Marc', sal = 100000

drop table salary
```

<u>Test Result</u>: According to the specifications, as in standard SQL

4.2. Table with *primary key* clause only

The table salary was created and a set of statements has been issued to perform the testing.

```
create table salary
(name char(10),
sal float,
time dateinterval not null
primary key (name, sal, time))


insert into salary values('John', 170000, '[1990-01-01, 1992-01-01)')
```

insert into salary values ('John', 2 * 100000, '[1992-01-01, 1994-01-01)')

insert into salary values ('John', 200000, '[1992-01-01, 1994-01-01)')

insert into salary (sal, time) values (170000, '[1990-01-01, 1992-01-01)')

insert into salary (sal, time) values (2 * 100000, '[1992-01-01, 1994-01-01)')

insert into salary values ('Marc', 170000, '[1990-01-01, 1992-01-01)', 'Marc', 2 * 100000, '[1992-01-01, 1994-01-01)')

insert into salary (select name, amount, days from s)

insert into salary(sal, time) select amount, time from s where amount = 14000

insert into salary(sal, time) select amount, days from s

delete from salary

insert into salary values('John', 170000, '[1990-01-01, 1992-01-01)')

insert into salary values('Marc', 170000, '[1990-01-01, 1992-01-01)')

insert into salary values ('Marc', 2 * 100000, '[1992-01-01, 1994-01-01)')

insert into salary select name, amount, days from s

insert into salary select * from s

delete from salary where sal = 10000

delete from salary where sal = 10000 or name = ''

delete from salary where interv(start(time), succ(now(), span(stop(time), start(time)))) cp
interv(stop(time), stop(merge(time, interv(start(time), now()))))

insert into salary values
('John', 170000, '[1990-01-01, 1992-01-01)')

delete from salary portion time = tointerv('[1990-01-01, 1991-01-01)')

delete from salary portion time = '[1990-01-01, 1991-01-01)'

delete from salary

insert into salary values ('John', 100000, '[1985-01-01, 1987-01-01)')

insert into salary values ('John', 130000, '[1987-01-01, 1989-01-01)')

insert into salary values ('John', 160000, '[1989-01-01, 1990-01-01)')

update salary set sal = 170000 where time = '[1989-01-01, 1990-01-01)' update salary set sal = 100000

update salary set time = '[1993-10-10, 1994-10-10)'

update salary set name = 'Jim', sal = 100

drop table salary

<u>Test Result</u>: According to the specifications

4.3. Table with *normalised* clause only

create table salary
(name char(10),
sal float,
time dateinterval not null
normalised (time))

insert into salary values
('John', 100000, '[1985-01-01, 1987-01-01)',
'John', 200000, '[1987-01-01, 1989-01-01)',
'John', 300000, '[1989-01-01, 1990-01-01)')

insert into salary values ('Marc', 100000, '[1985-01-01, 1987-01-01)')

insert into salary values ('John', 100000, '[1986-01-01, 1988-01-01)')

insert into salary values ('John', 300000, '[1989-11-01, 1998-01-01)')

delete from salary where name = 'Marc'

delete from salary portion time = '[1990-01-01, 1992-01-01)'
where stop(time) = '1998-01-01'

delete from salary portion time = '[1989-01-01, 1989-02-02)'
where start(time) = '1989-01-01'

delete from salary

insert into salary values ('John', 100000, '[1985-01-01, 1987-01-01)')

insert into salary values ('John', 200000, '[1986-01-01, 1988-01-01)')

insert into salary values ('John', 300000, '[1989-11-01, 1998-01-01)')

update salary set name = 'Marc'

update salary portion time = '[1988-11-01, 1988-01-10)'
set sal = 150000, time = '[1993-01-01, 1994-01-01)'

update salary portion time = '[1988-11-01, 1988-11-10)'
set sal = 150000, time = '[1993-01-01, 1994-01-01)'
where name = 'Marc'

update salary portion time = '[1988-11-01, 1988-01-10)'
set sal = 150000, time = '[1993-01-01, 1994-01-01)'
where name = 'John'

drop table salary

<u>Test Result</u>: According to the specifications

4.4. Table with both *primary key* and *normalised* clauses

create table salary
(name char(10),
sal float,
time dateinterval not null
normalised (time)
primary key(name, sal, time point))

insert into salary values
('John', 100000, '[1985-01-01, 1987-01-01)')

insert into salary values ('John', 130000, '[1987-01-01, 1989-01-01)')

insert into salary values ('John', 150000, '[1989-01-01, 1992-01-01)')

insert into salary values ('John', 150000, '[1990-01-01, 1992-01-01)')

insert into salary values ('John', 160000, '[1990-01-01, 1992-01-01)')

update salary portion time = '[1989-01-01, 1990-01-01)'
set sal = 130000
where sal = 150000

update salary portion time = '[1989-01-01, 1990-01-01)'
set sal = 130000, time = '[1988-01-01, 1990-01-01)
where sal = 150000

update salary portion time = '[1989-01-01, 1990-01-01)'
set sal = 120000, time = '[1988-01-01, 1990-01-01)
where sal = 150000

delete from salary portion time = '[1988-01-01, 1990-01-01)'
where sal = 150000

delete from salary portion time = '[1986-02-02, 1986-03-03)'
where sal = 100000

delete from salary

drop table salary

<u>Test Result</u>: According to the specifications

5. Evaluation of SELECT

For the evaluation of the SELECT statement, ten (10) broad categories of tests have been made. The categories contain tests that evaluate some part of the syntax of a VT-SQL SELECT statement. The first category contains very simple queries, having only a FROM clause. Each category of the tests, then, adds up to the complexity of the syntax of the queries, in order to test all aspects of the syntax of the VT-SQL SELECT statement.

5.1 SELECT <*list*> FROM <*tbl*>

- <list> contains all VT-SQL functions
- <list> contains all SQL functions
- <list> contains combination of VT-SQL functions (not exhaustive)
- <list> contains combination of SQL functions (not exhaustive)
- <list> contains combination of SQL and VT-SQL functions (not exhaustive)
- <list> contains '*'

<u>Test Result</u>: According to the specifications

5.2 SELECT A=<*list*> FROM <*tbl*>

- <list> contains all VT-SQL functions
- <list> contains all SQL functions
- <list> contains combination of VT-SQL functions (not exhaustive)
- <list> contains combination of SQL functions (not exhaustive)
- <list> contains combination of SQL and VT-SQL functions (not exhaustive)

<u>Test Result</u>: According to the specifications

5.3 SELECT <*list*> FROM <*tbl*> WHERE <*condition*>

- <condition> contains all VT-SQL functions
- <condition> contains all SQL functions
- <condition> contains combination of VT-SQL and SQL functions with *and, or, not* (not exhaustive)
- <condition> contains combination of SQL functions (not exhaustive)
- <condition> contains combination of VT-SQL functions and VT-SQL predicates (not exhaustive)

<u>Test Result</u>: According to the specifications

5.4 SELECT * FROM <*tbl*> GROUP BY <*group_list*> HAVING <*hav_condition*>

- <group_list> contains combination of column names
- <hav_condition> contains all SQL functions

- < hav_condition > contains a combination of VT-SQL and SQL functions with *and*, *or, not* (not exhaustive)
- < hav_condition> contains combination of VT-SQL functions and VT-SQL predicates (not exhaustive)

Test Result: According to the specifications

5.5 SELECT * FROM <*tbl*> REFORMAT AS <*ref_clause*>

- <ref_clause> contains UNFOLD and all combinations of column names or numbers referring to column names for single, or two tables
- <ref_clause> contains UNFOLD ALL and all combinations of column names or numbers referring to column names for single, or two tables
- <ref_clause> contains FOLD and all combinations of column names or numbers referring to column names for single, or two tables
- <ref_clause> contains FOLD ALL and all combinations of column names or numbers referring to column names for single, or two tables
- <ref_clause> contains UNFOLD and FOLD and all combinations of column names or numbers referring to column names for single, or two tables

Test Result: According to the specifications

5.6 SELECT * FROM <*tbl*> NORMALISE ON <*nor_list*>

- <nor_list> contains all names or numbers referring to column names for single, or two tables

Test Result: According to the specifications

5.7 <*query1*> WHERE <*arg1*> <*op*> <*query2*>

- <arg> contains column names or VT-SQL functions (not exhaustive)
- <arg> is of type *date* or *dateinterval* and <op> is one of <, >, =, precedes, follows, prequals, folequals
- all, any, exist before <*query2*>

Test Result: According to the specifications

5.8 <*query1*> UNION <*query2*>

- <query1> does not have REFORMAT or NORMALISE and <query2> does not have REFORMAT or NORMALISE
- <query1> has REFORMAT or NORMALISE and <query2> does not have REFORMAT or NORMALISE
- <query1> does not have REFORMAT or NORMALISE and <query2> has REFORMAT or NORMALISE
- <query1> has REFORMAT or NORMALISE and <query2> has REFORMAT or NORMALISE

Test Result: According to the specifications

5.9 *<query1>* EXCEPT *<query2>*

- <query1> does not have REFORMAT or NORMALISE and <query2> does not have REFORMAT or NORMALISE
- <query1> has REFORMAT or NORMALISE and <query2> does not have REFORMAT or NORMALISE
- <query1> does not have REFORMAT or NORMALISE and <query2> has REFORMAT or NORMALISE
- <query1> has REFORMAT or NORMALISE and <query2> has REFORMAT or NORMALISE

Test Result: According to the specifications

REFERENCES

[01P 93a] 01PLIROFORIKI 'Specification of system requirements', ORES Deliverable C2, Athens, 1993.

[01P 93b] 01PLIROFORIKI 'Specification of Valid Time Formalism', ORES Deliverable C3, Athens, 1993.

[01P 93c] 01PLIROFORIKI 'Implementation of Valid Time Algebra', ORES Deliverable C4, Athens, 1993.

[01P 93d] 01PLIROFORIKI 'Specification of Valid Time SQL, Revision I', ORES Deliverable D2, Athens, 1993.

[01P 93e] 01PLIROFORIKI 'Design of Valid Time SQL', ORES Deliverable D3, Athens, 1993.