Transaction Support in a Temporal DBMS

Costas Vassilakis
Department of Informatics, University of Athens
Panepistimiopolis, TYPA buildings, 15771, Athens, Greece

Nikos Lorentzos
Agricultural University of Athens
Iera Odos 75, 11855, Athens, Greece

Panagiotis Georgiadis
Department of Informatics, University of Athens
Panepistimiopolis, TYPA buildings, 15771, Athens, Greece

Abstract

Transactions are a significant concept in database systems, facilitating functions both at user and system level. However transaction support in temporal DBMSs has not yet received enough research attention. In this paper, we present techniques for incorporating transaction support in a temporal DBMS, which is implemented as an additional layer to a commercial RDBMS. These techniques overcome certain limitations imposed by the underlying RDBMS, and avoid excessive increment of the log size.

1. Introduction.

Transactions are an important feature of database systems. At user level, transactions are the unit of *integrity* ([1]), allowing the database to go from a valid state to another passing from an invalid state, provide the ability to undo erroneous changes, and provide an atomic, "all-or-nothing" abstraction ([2]), which makes programming tasks easier. At system level transactions constitute the unit of *sharing* ([1]), and *recovery*. For example, locks acquired during a session are released at the end of the active transaction and after a system failure the database can be restored to its state, at a COMMIT point.

In this paper, we present techniques for incorporating transactions in a layered temporal DBMS ([3]). The temporal DBMS uses an interval extended relational model ([4]), for temporal data representation. VT-SQL ([5]), a consistent extension to SQL89, is its data definition and manipulation language. Data are timestamped at tuple level, and valid time relations are coalesced. The temporal DBMS is split in two layers, the lower one being a commercial RDBMS (INGRES), which is used for data storage and retrieval. The kernel of the RDBMS has been extended, to support an additional data type, DATEINTERVAL, as well as operations on this type. The upper layer is the *temporal engine*, coded in C and embedded SQL, which supports the valid time semantics. One component in this layer is VT-RA, a Valid Time Relational Algebra. Figure 1 illustrates the overall temporal RDBMS architecture.

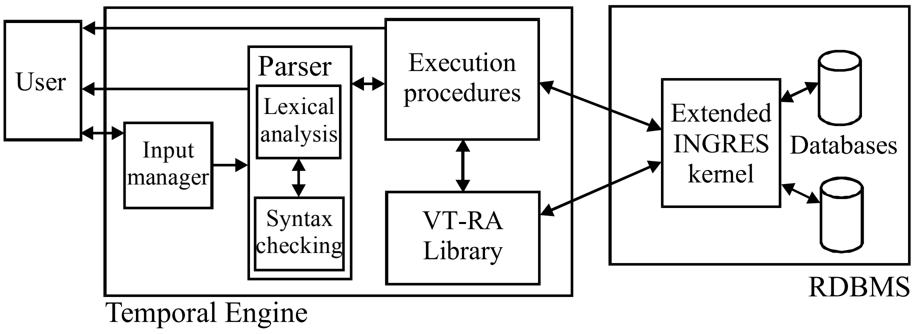

Figure 1 - Temporal RDBMS architecture

Transaction support in such a layered temporal RDBMS cannot rely on the support offered by the RDBMS for two reasons:

1. In many cases, the temporal engine must create temporary tables to store intermediate results (e.g. the table holding the updated tuples in the execution of the UPDATE statement or the table holding the result of the extended SELECT, in the execution of the INSERT statement, according to the algorithms presented below). However, in some DBMSs (e.g. ORACLE), issuing a DDL statement, such as CREATE TABLE or DROP TABLE, introduces an implicit commit point ([6]), so changes to the database state made before that point, cannot be undone using a ROLLBACK statement. Other DBMSs (e.g. Sybase) disallow the usage of DDL statements within multi statement transactions ([7]).
2. Writing results to intermediate tables is logged (as all modification operations are). Since, the execution procedures of the VT-SQL statements produce substantial amount of intermediate results, the size of the log space increases considerably. Thus, techniques should be developed to reduce log space requirements.

The rest of this paper is organised as follows: In section 2, the Valid Time Relational Algebra is presented, in brief. In section 3, the syntax and semantics of VT-SQL are described. Section 4 presents the algorithms used for the DML statements, in order to provide transaction support. Section 5 addresses protection and crash recovery issues arising from the techniques described in section 4. The last section concludes and outlines future work.

2. Valid time relational algebra.

The design of the temporal RDBMS has been based on Valid Time Relational Algebra (VT-AL) ([4]), a consistent extension to Codd's algebra ([8]). New data types have been introduced for the representation of time, namely DATE and DATEINTERVAL. Date literals have the format YYYY-MM-DD, which is a more readable form of the ANSI standard. DATEINTERVAL values are denoted as *[d_i, d_j)*, where d_i and d_j are dates, and d_i is before d_j. A DATEINTERVAL value contains all dates from d_i and up to, but not including d_j. VT-RA defines transformations between the representations of time (points and intervals), new predicates for interval comparison, as well as extended UNION and EXCEPT operations, which are applicable to relations containing attributes of type DATE and/or DATEINTERVAL. These operations are described briefly in the following paragraphs. A detailed presentation can be found in [4] and [9].

2.1 Fold.

Let R be a relation whose schema is (A_1, A_2, ..., A_n). When R is folded on column A_i (denoted as FOLD[A_i] (R)), where the domain of A_i is of a DATE or DATEINTERVAL type, all its tuples whose A_j columns have identical values $\forall j \neq i$, and their A_i columns can merge (i.e. are overlapping or adjacent), are replaced in the resulting relation by a single tuple with the same values in the A_j columns$\forall j \neq i$, but its *i*-th component is formed by the *merging* of the *i*-th components of these tuples. For example, if SALARY is any of the relations in figure 2, then FOLD[Time] (SALARY) yields the relation in figure 3.

The FOLD operation may apply to multiple columns. This is denoted by FOLD [A_{i_1}, A_{i_2}, ..., A_{i_n}] (R), and is equivalent to folding relation R on column A_{i_1}, then on column A_{i_2} and so on up to column A_{i_n}.

SALARY

Name	Amount	Time
John	10K	d1
John	10K	d2
...	...	...
John	10K	d4
John	10K	d10
...	...	...
John	10K	d14
Alex	12K	d1
...	...	...
Alex	12K	d19

(a)

SALARY

Name	Amount	Time
John	10K	[d1, d3)
John	10K	[d2, d5)
John	10K	[d10, d15)
Alex	12K	[d1, d10)
Alex	12K	[d10, d15)
Alex	12K	[d15, d18)
Alex	12K	[d16, d20)

(b)

Figure 2 - Two valid time relations

SALARY

Name	Amount	Time
John	10K	[d1, d5)
John	10K	[d10, d15)
Alex	12K	[d1, d20)

Figure 3 - A valid time relation

2.2 Unfold.

When a relation R is unfolded on attribute A_i (denoted as UNFOLD [A_i] (R)), where the domain of A_i is of a DATE or DATEINTERVAL type, each tuple $(t_1, ..., t_{i-1}, t_i, t_{i+1}, ..., t_n)$ of R is replaced in the resulting relation by a family of tuples $(t_1, ..., t_{i-1}, t_{i_j}, t_{i+1}, ..., t_n)$, where each t_{i_j} is a date included in t_i (as a trivial case, a date is considered to include only itself). For example, if SALARY is the relation in figure 3, then UNFOLD[Time] (SALARY) yields the relation in figure 2(a). An UNFOLD may apply to multiple columns; this is denoted by UNFOLD [A_{i_1}, A_{i_2}, ..., A_{i_n}] (R), and is equivalent to unfolding relation R on column A_{i_1}, then on column A_{i_2} and so on up to column A_{i_n}.

2.3 Normalise.

The NORMALISE operation can be applied to multiple columns of type DATE or DATEINTERVAL. It is denoted by NORMALISE [A_{i_1}, ..., A_{i_n}] (R), and is semantically equivalent to FOLD [A_{i_1}, ..., A_{i_n}] (UNFOLD [A_{i_1}, ..., A_{i_n}] (R)). The NORMALISE operation is thus introduced for notational convenience.

2.4 PUnion.

The PUNION operation can be applied to two union-compatible relations and operates on multiple columns of type DATE or DATEINTERVAL. Two relations R and S are union-compatible if:

1. The number of columns in R is the same as the number of columns in S and
2. Column R_i is type-compatible with column S_i, $\forall$ i.

The PUNION operation of two relations R and S is denoted by R PUNION [A_{i_1}, ..., A_{i_n}] S and is equivalent to applying NORMALISE [A_{i_1}, ..., A_{i_n}] to the result of R UNION S. For example, if a relation S has a single tuple,

(John, 10K, [d3, d12))

and SALARY is the table in figure 3, then the result of SALARY PUNION [Time] S consists of the two tuples

(John, 10K, [d1, d15))
(Alex, 12K, [d1, d20))

2.5 PExcept.

The PEXCEPT operation can be applied to two union-compatible relations and operate on multiple columns of type DATE or DATEINTERVAL. The PEXCEPT operation of two relations R and S is denoted by R PEXCEPT [A_{i_1}, ..., A_{i_n}] S and is equivalent to applying the FOLD [A_{i_1}, ..., A_{i_n}] operation to the result of the

UNFOLD [A_{i_1}, ..., A_{i_n}] (R) EXCEPT UNFOLD [A_{i_1}, ..., A_{i_n}] (S)

operation. For example, if a relation S consists of the two tuples

(John, 10K, [d10, d15))
(Alex, 12K, [d10, d15))

and SALARY is the table in figure 3, then SALARY PEXCEPT [Time] S consists of the tuples

(John, 10K, [d1, d5))
(Alex, 12K, [d1, d10))
(Alex, 12K, [d15, d20))

Tables which require the use of PUNION (PEXCEPT) for data insertion (deletion) rather than UNION (EXCEPT) are called *normalised.*

3. VT-SQL syntax and semantics.

The DDL statements of VT-SQL are the same as in SQL, except for the CREATE TABLE statement, which has been extended to allow for the specification of the primary key of a valid time table (see section 3.2, below). In the following paragraphs, therefore, the syntax and semantics of the VT-SQL DML statements is presented. A complete presentation of the VT-SQL syntax and semantics, can be found in [5]. In the syntax, which follows, terms enclosed in brackets ([]) are optional; braces ({}) are used for items that may be repeated zero or more times; parentheses are used for grouping, and single quotes are used for parentheses which must be typed literally; capitals indicate reserved words and italics are used for user-provided values.

3.1 The Select statement.

The VT-SQL syntax for the SELECT statement is

```
extended-select
[(UNION | UNION ALL | EXCEPT) [ResultColumnList]
[extended-select]
[ORDER BY ResultColumn [ASC | DESC]
          {, ResultColumn [ASC | DESC]}]
```

The *extended-select* is defined as

```
sql-select
[REFORMAT AS [(FOLD | UNFOLD) columnList
                 {(FOLD | UNFOLD) columnList}]
[NORMALISE ON ResultColumnList]
```

(Note than another version of UNFOLD, namely UNFOLD ALL, is also described in [9] but has been omitted here, for brevity reasons.) An *extended-select* is executed by evaluating its *sql-select* part, and then applying the REFORMAT/NORMALISE operations specified by the corresponding clauses. If the VT-SQL select statement includes a second *extended-select*, then the result of each of the two *extended-select* is computed, and a VT-RA operation is applied to them, in order to evaluate the final query outcome (this presumes that the schemata of the results of the two *extended-selects* are union-compatible). The UNION keyword specifies that either the VT-RA PUNION or the standard UNION operation should be applied, depending on whether the keyword UNION is followed by a column list or not, respectively. In the former case, the column list specifies the columns on which the PUNION operation will normalise the final result; these columns must be of type DATEINTERVAL or DATE. The ALL keyword may follow the UNION keyword, indicating that duplicate occurrences of result tuples should be retained. Analogously, the EXCEPT keyword specifies that either the PEXCEPT or EXCEPT operation must be applied, depending on whether the keyword is followed by a column list or not. Finally, the ORDER BY clause follows the SQL89 specification.

3.2 The Insert statement.

The syntax of the VT-SQL INSERT statement is identical with that of the SQL INSERT, except one case: If the tuples to be inserted are specified by a query, this query may be an *extended-select*. When data are inserted in a non-normalised table, the semantics of the INSERT statement are identical with the semantics of its SQL counterpart. If, however, the tuples are inserted into a normalised table, then an ordinary insertion takes place, followed by a NORMALISE operation on the appropriate columns.

The concept of the key of a table has been extended to normalised relations. Thus, the key of SALARY, in figure 3, is <Name, Time-*p*>. This means that

SALARY may never contain two tuples, *t1* and *t2*, which satisfy both (i) t1.Name = t2.Name and (ii) t1.Time and t2.Time are two dateintervals which have at least one date in common. We say that SALARY preserves the uniqueness of the primary key *at a date level*.

If the key of a normalised relation R has been defined and a piece of data, which is to be inserted, has already been recorded in R, then the transaction is aborted. For example, consider SALARY, in figure 3, with key <Name, Time-*p*>. If we issue the command

```
INSERT INTO SALARY
VALUES ('John', 10K, '[d4, d10)')
```

the insertion will fail, because John's salary for date *d4* is already in SALARY. This implementation convention is an extension of that in standard SQL.

3.3 The Delete statement.

A PORTION clause has been added to the DELETE statement, which may be used when deleting data from a normalised table. If the PORTION is present in a DELETE statement, it designates the valid time period to which the deletion applies. If the PORTION clause is not specified, the deletion applies to whole tuples. For example, after the execution of the command

```
DELETE FROM SALARY
PORTION Time = '[d3, d12)'
WHERE Name = 'John'
```

then SALARY, in figure 3, will consist of the tuples

(John, 10K, [d1, d3))
(John, 10K, [d12, d15))
(Alex, 12K, [d1, d20))

3.4 The Update statement.

When updating a non-normalised table, UPDATE behaves exactly as in SQL. If, however, the table is normalised, the update is followed by a NORMALISE operation on the appropriate columns. Analogously to the DELETE statement, the UPDATE of a normalised table may contain a PORTION clause, which specifies the valid time period to which the update is applied. For example, the command

```
UPDATE SALARY
PORTION Time = '[d3, d5)'
SET Name = 'Tom'
WHERE Name = 'John'
```

will result in that SALARY, in figure 3, will consist of the tuples

(John, 10K, [d1, d3))
(John, 10K, [d10, d15))
(Tom, 10K, [d3, d5))
(Alex, 12K, [d1, d20))

Again, if the table preserves the uniqueness of the primary key *at a date level*, the transaction is aborted if the update may result in a table, which violates this uniqueness.

4. Transaction support.

Transaction support can be facilitated in a layered temporal RDBMS by having the temporal engine connected *twice* to the RDBMS, thus opening two sessions, the *user session* and the *system session*. User tables are always accessed (both for reading and writing) through the user session. The system session is used in order to issue to the RDBMS the DDL statements which create or drop the temporary tables, as well as the DML statements which insert or modify data in these tables. Data in the temporary tables can be read by both the user and the system session.

Different sessions to the RDBMS have independent commit modes, i.e. statements issued through a session do not affect the commit status of other sessions. Locking also takes place at session level, i.e. objects locked in shared or exclusive mode by one session are not available for update or access to other sessions, so care must be taken that the locking scheme does not lead to deadlocks.

In the following paragraphs, the usage of the two sessions, as well as the locking schemes employed, to provide full transaction support, are described in detail.

4.1 The Select statement.

Three cases are considered for the evaluation of the SELECT statement (for a complete description, see [3]):

1. If the SELECT statement does not imply the application of operations not supported by the RDBMS (i.e. REFORMAT, NORMALISE, PUNION and PEXCEPT operations), the temporal engine opens a cursor through the user session, for the user query. The result tuples are fetched through this cursor and presented to the user.
2. If the user query consists of only one *extended-select* then the following steps are taken:

A. A temporary table is created through the system session, whose schema matches the schema of the table resulting from the *sql-select*. A cursor is opened through the user session for the *sql-select* part of the user query.
B. The cursor opened in step (A) is used to fetch the result tuples, which are inserted into the temporary table through the system session. When all result tuples have been inserted into the temporary table, the system session commits, emptying its log space.
C. The REFORMAT and NORMALISE clauses are executed from within the system session. As soon as each operation stated in each clause completes, the temporary table holding the results of the previous step is dropped through the system session, and the system session commits.
D. The tuples contained in the final intermediate table are fetched through the system session, and forwarded to the user. When all data has been exhausted, the final intermediate table is dropped through the system session, and the system session commits.

3. If the user query consists of two *extended-select* statements, combined by a UNION, EXCEPT, PUNION, PUNION ALL or PEXCEPT operation, then the following procedure is used:
   A. Steps (A)-(C) of case 2 are performed to evaluate each of the *extended-select* statements, storing the results in intermediate tables.
   B. The UNION, EXCEPT, PUNION, PUNION ALL or PEXCEPT operation is applied through the system session to the intermediate tables produced in step (A), and the results are stored in a temporary table. Upon operation completion, the intermediate tables produced in step (A) are dropped through the system session, and the system session commits.
   C. The tuples contained in the temporary table created in step (B) are fetched through the system session, and forwarded to the user. Finally, the temporary table is dropped through the system session, and the system session commits.

The execution procedure described above for the SELECT statement does not introduce any implicit commit points for the user session and does not expand the user session log file (only the system session's log is expanded, but only transiently, as it is truncated at the system session's commit points). Furthermore, since the user tables are handled by the user session and all temporary tables are accessed through the system session, no deadlock problems are introduced.

4.2 The Insert statement.

Insertions issued against non-normalised tables, can directly be forwarded for execution, through the user session, to the underlying RDBMS. The algorithm employed for data insertion in a normalised table *R* considers four distinct cases ([10]), (i)-(iv), below. In the sequel, the columns of any table *R* will be denoted as $R_{c1}, R_{c2}, \ldots, R_{cN}, R_{vt}$, with R_{vt} being the column storing the valid time of the tuple. Columns not participating in the primary key -which includes all the columns from R_{c1} through R_{cN} if no primary key is defined on the table- will be denoted as R_{nonKey}, and columns participating in the primary key -except for R_{vt}- will be referenced as R_{key}.

Case (i): *The values to be inserted are specified by means of the* VALUES *clause, and no primary key is defined on the table.*

The temporal engine opens a cursor *ins_cur* on the target relation through the user session, selecting all tuples which are *value equivalent* to the insertion tuple (i.e. each column in R_{nonKey} is equal to the corresponding column in the insertion tuple) and have overlapping or adjacent valid times. Each qualifying tuple is deleted from the target relation through cursor *ins_cur* (for which the fetched tuple is current), and the valid time of the insertion tuple is replaced by the union of its former value and the timestamp of the deleted tuple (the union of two adjacent or overlapping dateintervals is a dateinterval containing all time points in both arguments). Finally, the insertion tuple is appended to the target relation, through the user session.

In this case, all interaction with the RDBMS is performed through the user session, so no deadlock problems are introduced. Furthermore, changes to the database state are almost minimal, so log size increment is kept low (the minimum changes would be deleting all the selected value equivalent tuples except the last one, whose timestamp should be updated to the union of the timestamps of all the selected value equivalent tuples. However, cursors provide no indication whether the current tuple is the last or not, so this approach would require a second scan of the table, which would penalise the performance unacceptably).

Case (ii): *The values to be inserted are specified by means of the* VALUES *clause, and a primary key has been defined on the table*.

The algorithm starts off with one insertion tuple, holding the values specified in the VALUES clause. A *savepoint Save1* is created for the user session (the name of the savepoint is actually a unique, system-generated string), and a cursor *ins_cur* is opened on the target table, through the user session, selecting all tuples for which all the columns in R_{key} have values equal to the values of the corresponding table in the insertion tuple, and their valid times are adjacent or overlapping to the timestamp of the insertion tuple. For each tuple that is fetched through the cursor, the following checks are made:

1. If the valid time of the fetched tuple is overlapping with the valid time of the insertion tuple, then the INSERT statement violates primary key uniqueness. The user session is rolled back to the savepoint created at the beginning of the algorithm, by issuing a

   ```
   ROLLBACK TO Save1
   ```

   statement through the user session, and further processing is aborted.
2. If all columns in R_{nonKey} of the fetched tuple have values equal to the corresponding columns of the insertion tuple, then the fetched tuple is deleted from the target table, through cursor *ins_cur*, and the valid time of the insertion tuple is replaced by the union of its former value and the value of the fetched tuple's valid time.

3. In all other cases, i.e. if any column in R_{nonKey} of the fetched tuple is not equal to the corresponding column of the insertion tuple, the algorithm continues with the next tuple.

   When no more tuples can be fetched through the cursor, the insertion tuple is appended to the table, through the user session.

Remarks about locking problems and log size increment for the previous case (in which no primary key is defined), hold for this case too.

Case (iii): *The values to be inserted are specified by means of an extended-select query, and no primary key has been defined on the table.*

The *extended-select* is evaluated as described in paragraph 4.1, and the result is stored in a temporary table (if the query can directly be supported by the RDBMS, a cursor is opened through the user session, fetching the result tuples, which are stored in a temporary table, through the system session; if the query consists of an SQL SELECT, followed by a REFORMAT and/or a NORMALISE clause, then only steps (A) to (C) are performed). The temporary table holding the results of the *extended-select* will be denoted as *T1* (its name is actually a unique, system-generated string). Subsequently, the temporal engine opens a cursor on table *R* through the user session, selecting all tuples which can be coalesced with any tuple in *T1*, i.e. tuples for which every column in R_{nonKey} has value equal to the corresponding column in $T1_{nonkey}$ and the value of R_{vt} is overlapping or adjacent to the value of $T1_{vt}$. Since tuple fetching through this cursor implies access to table *T1*, which will be dropped afterwards through the system session, it is important that this access does not place any locks on the tuples of *T1*. This is accomplished by issuing a statement

```
SET LOCKMODE ON TABLE T1 WHERE READLOCK = NOLOCK;
```

through the user session, prior to opening the cursor (this command is an INGRES extension to SQL89 ([11]); the syntax in other DBMSs may be different). Each qualifying tuple is fetched into memory, and deleted from the target table, through the user session, and subsequently inserted through the system session in table *T1*. Afterwards, the system session is used to perform a normalisation operation on table *T1*, each tuple of the FOLD operation's result is fetched into memory through the system session, and inserted into the target table through the user session. Finally, the temporary tables are dropped through the system session, and the system session commits.

This algorithm does not present deadlock problems, since when the user session accesses the temporary tables, created and altered through the system session, it does so without requiring or imposing any locks. During these accesses, the system session is quiescent, so no concurrency problems are introduced, due to absence of locks. Finally, the only changes made through the user session are the deletions of the tuples that can be coalesced with any of the insertion tuples, plus the actual insertion of the resulting tuples in the target table, keeping log size increment at reasonable levels.

Case (iv): *The values to be inserted are specified by means of an extended-select query, and a primary key has been defined on the table*.

A *savepoint* is introduced for the user session, the *extended-select* is evaluated and the results are stored in a temporary table, as described for the previous case. Afterwards, the temporal engine renounces its locking rights for the user session on table *T1* by issuing a statement

```
SET LOCKMODE ON TABLE T1 WHERE READLOCK = NOLOCK;
```

through the user session; the same session is used to open a cursor, on the join of the target table and *T1*, selecting those tuples for which all columns in R_{Key} have values equal to the corresponding columns of table *T1* and R_{vt} is overlapping or adjacent to $T1_{vt}$. For each one of these tuples, all columns of table *R*, and columns in $T1_{nonKey}$ and $T1_{vt}$ are fetched into the main memory, and the following checks are made:

1. If the values of R_{vt} and $T1_{vt}$ are overlapping, the insert operation is aborted, due to the presence of duplicate keys; the database is rolled back to the savepoint introduced at the beginning of the algorithm execution, and further processing is aborted.
2. If the values of R_{vt} and $T1_{vt}$ are adjacent, and all columns in R_{nonKey} have values equal to the corresponding columns in $T1_{nonKey}$, the current tuple of the target table is deleted, through the user session, and subsequently inserted into *T1*, through the system session (the values are currently into the main memory, so they can be inserted immediately).
3. In all other cases, the fetched tuple is ignored.

When this process completes, the system session commits, and a cursor is opened on table *T1*, through the system session, fetching all the fields of each row, sorted on columns $T1_{key}$, $T1_{vt}$, $T1_{nonkey}$, in that order. The first row is fetched and marked as *working tuple*, and the algorithm proceeds as follows:

1. The next tuple is fetched through the cursor. If data has been exhausted, *working tuple* is inserted into the target table through the user session and the algorithm continues with step (5), otherwise the fetched tuple is marked as *current tuple*, and the algorithm continues with step (2).
2. If the value of any of the columns in $T1_{key}$ is different in *working tuple* and *current tuple*, or the values of $T1_{vt}$ in the two tuples are neither overlapping nor adjacent, then *working tuple* is inserted into the target table through the user session, *current tuple* replaces *working tuple* and step (1) is performed again.
3. If the values of $T1_{vt}$ in *working tuple* and *current tuple* are overlapping, then the operation produces duplicate keys; the database is rolled back to the savepoint introduced at the beginning of the algorithm execution and further processing is aborted.
4. If the value of any of the columns in $T1_{nonKey}$ is different in *working tuple* and *current tuple*, then *working tuple* is inserted into the target table through the user session and *current tuple* replaces *working tuple*; otherwise, $T1_{vt}$ in working tuple is replaced by the union of its former value and the value of $T1_{vt}$ in the *current tuple* and *current tuple* is discarded and control passes to step (1).

5. Temporary tables are dropped through the system session and the system session commits.

The remarks made about the absence of deadlocks and log size increment for the previous case, hold for this case too.

4.3 The Delete statement.

If the PORTION clause is not specified in the DELETE statement, the request is directly forwarded to the underlying RDBMS for execution, through the user session. If, however, the PORTION clause is specified, the following actions are taken, in order to satisfy the user request: The temporal engine opens, through the user session, a cursor on the target table, selecting the rows matching the criteria stated in the WHERE clause and having valid times overlapping with the period specified in the PORTION clause (denoted as *period*, hereafter). For each selected tuple, the values of all fields along with the value of *period* are fetched into main memory, and one of the following actions is taken:

1. If the value of *period* is a superinterval of the value of R_{vt}, then the tuple is deleted from the target table, through the user session.
2. If the difference R_{vt} - *period* yields exactly one interval (i.e. the time points included in R_{vt} but not in *period* are consecutive and, consequently, can be represented by a single DATEINTERVAL value) then the value of R_{vt} of the current tuple is set to R_{vt} - *period*, through the user session.
3. If the difference R_{vt} - *period* yields two intervals, *diff1* and *diff2*, the value of R_{vt} of the current tuple is set to *diff1*, and a new tuple is appended to the target table, for which columns R_{c1}, ..., R_{cN} are equal to the values of the corresponding columns in the current tuple, whereas the value of column R_{vt} is equal to *diff2*. Both the value change and the tuple insertion are performed through the user session.

Since the whole interaction is performed through the user session, the algorithm is deadlock free. The changes made to the database state are also kept to an absolute minimum, resulting in the minimum increment to the log size.

4.4 The Update statement.

If the UPDATE statement is applied to a non-normalised table, the UPDATE can directly be forwarded for execution to the underlying DBMS. If, however, the target table is normalised, then the following cases are considered:

Case (i): *The table has no primary key, and the PORTION clause is not specified.*

The temporal engine opens a cursor on the target table, through the user session, selecting tuples qualifying with respect to the WHERE clause. For each selected tuple, the updated values of the fields changed by the SET clause, rather than the original values are fetched; the current tuple is deleted from the table, through the user session, and a tuple containing the updated values is stored in a temporary table, through the system session (the table will be denoted as *update_temp* and is created through the system session). When all qualifying tuples have been fetched, the system session commits, and the algorithm described for case (iii) in paragraph 4.2 is employed to insert the tuples in *update_temp* into the target table (obviously, the step involving the execution of the *extended-select* is not performed; table *T1* mentioned in paragraph 4.2 is actually the *update_temp* table, produced in the previous step).

No deadlock problems are introduced through this algorithm, since the user session is used to access the user table, and subsequently a deadlock-free insertion algorithm is used. Log size increment is kept low, since every tuple update maps to one deletion and one insertion through the user session (actually, more deletions can performed, if some updated tuple can be coalesced with some tuple which is not updated).

Case (ii): *The table has no primary key and the PORTION clause is specified.*

The temporal engine opens a cursor on the target table through the user session, selecting the tuples which qualify with respect to the WHERE clause and for which the value of R_{vt} overlaps with the value of the period specified in the PORTION clause. For each qualifying tuple, all the original values of the columns, the new values for the columns to be updated and the value of the period in the PORTION clause are fetched into main memory, the part of the tuple corresponding to the period specified in the PORTION clause expression is deleted from the target table (following the optimised algorithm of the DELETE statement), and a tuple is inserted through the system session into a temporary table *update_temp* (which will have been created through the system session). The values of the columns of this tuples are determined using the following algorithm:

1. If the column appears on the left hand side of an assignment in the SET clause, then the value of the corresponding right hand side expression is used.
2. If the column is not updated, then its original value is used, except for column R_{vt}, for which the value of the *expression* appearing in the PORTION clause is used.

When all qualifying rows have been dealt with, the rows in *update_temp* are inserted in the target table using the algorithm described in paragraph 4.2 for case (iii). (The step of evaluating the *extended-select* is skipped and *update_temp* replaces *T1*.)

Remarks made on the absence of deadlocks for the previous case, hold for this case too. Log size increment is also kept low, with every update mapping to either one tuple deletion and one insertion, or one update and one insertion or one update and two insertions, depending on the portion of the tuple which will be updated.

Case (iii): *The table has a primary key and the PORTION clause is not specified.*

The algorithm employed for case (i) can be used here, modified so that a *savepoint* is introduced for the user session at the begining of the operation, and the resulting tuples are inserted into the target table using the algorithm for inserting data in a table for which a key is defined (case (iv) of the INSERT statement, with the necessary amendments: The step of evaluating the *extended-select* is skipped, *update_temp* replaces *T1* and the savepoint introduced at the start of the operation is used when the database should be rolled back due to primary key uniqueness violation). Remarks on the absence of deadlocks and log size increment for case (i) hold for this case too.

Case (iv): *the table has a primary key and the PORTION clause is specified.*

The algorithm described for case (ii) can be used for this case, modified so that a *savepoint* is introduced for the user session at the begining of the operation, and the resulting tuples are inserted in the target table using the algorithm for inserting data in a table for which a key is defined (case (iv) of the INSERT statement, with the necessary amendments). Remarks on the absence of deadlocks and log size increment for case (ii) hold for this case too.

5. Protection and crash recovery.

This section presents techniques for dealing with protection and crash recovery issues, arrising from the algorithms presented in section 4. A protection scheme which prevents ad-hoc modification of temporary tables is presented in section 5.1, and an algorithm for removing temporary tables which remained in the database because of a system crash is described in section 5.2.

5.1 Protection scheme for temporary tables.

The correctness of data contained in temporary tables is a crucial point for the successful completion of the operations described in the previous sections. Thus, it is important to prevent users from modifying the contents of temporary tables. User access to these tables must also be avoided, as locks may be placed, which can lead to substantial delays (e.g. the system session will have to wait until these locks are released, before dropping the table) or even deadlocks.

In general, the life span of temporary tables is limited and, actually, users ignore the names of temporary tables, so the probability of user access is limited. However, it is possible that during the evaluation of a complex query or in a period of increased system load, some user acquires information about the name of a temporary table (by querying an RDBMS system catalogue) and access it or modifies it, by issuing a query. Temporary tables can be protected from unauthorised access using the following techniques:

1. A special user id, e.g. *temporal*, is created in the RDBMS, and the CREATE TABLE privilege is granted to this user id for every database handled by the RDBMS. The system session is opened under this special user id, using the IDENTIFIED BY clause of the embedded SQL connect statement.
2. If a temporary table must be accessed through the user session, this access is preceded by the command sequence

   ```
   GRANT SELECT ON TempTable TO UserName
   COMMIT
   LOCK TABLE TempTable IN EXCLUSIVE MODE
   ```

   which is issued through the system session. (If a LOCK TABLE statement is not available, the same effect can be accomplished by setting the locking granularity of the system session to table level, its read access locking mode to exclusive and accessing a single tuple of the temporary table through it).

TempTable is the name of the temporary table which must be accessed by the user session, and *UserName* is the user id under which the user session is opened. The SELECT privilege is revoked as soon as the user session completes the necessary access, by issuing the statements

```
REVOKE SELECT ON TempTable FROM UserName
COMMIT
```

through the system session.

3. After each COMMIT point of the system session, all existent temporary tables, on which the SELECT privilege is granted, are locked in exclusive mode.

Using a different user id for the system session protects the temporary tables from unauthorised modifications, since none of the INSERT, DELETE and UPDATE privileges are granted by default to user ids different than the table creator. Read access to the temporary tables is limited to the period during which it is absolutely necessary and, when it is permitted, the table is locked in exclusive mode by the system session. Thus, this access has to be done in the "no lock" fashion employed by the user session, which eliminates any delay or deadlock possibility.

5.2 Removing remnant temporary tables.

It is possible that, during the execution of a query, requiring the creation of intermediate tables, either some temporal engine, or the RDBMS, or the computer system, on which any of these programs are executed, crashes. Since the system session commits its results with temporary tables present in the database, the RDBMS considers these tables permanent and will try to preserve them through system crashes. Therefore, a method must be provided, to remove these tables from the database. One of the following two approaches can be followed:

1. A *naming convention* can be adopted for the temporary tables, e.g. their name should always start with the string `tt_temp`. When the RDBMS recovers from a crash, a program can be invoked by the database administrator, which drops all tables whose name starts with this specified string (the names can be determined by querying the RDBMS system catalogues). Users should be warned about this policy, so they will not create a table which might be removed by this procedure.
2. A special table can be introduced, which can be used by the temporal engine to store the names of the temporary tables currently present in the database. A temporary table should then be registered in this catalogue *before* the CREATE TABLE statement, which creates this temporary table, is issued to the RDBMS. The registration should be removed only after the corresponding DROP TABLE statement has been issued (data insertion and deletion in the catalogue is performed through the system session). Upon crash recovery, a program is invoked, which removes from the database all tables registered in this catalogue.

6. Conclusions - future work.

In this paper, we presented techniques for supporting transactions in a layered temporal DBMS. The techniques presented exploits the transaction support features of the underlying RDBMS, using a second connection to it, through which operations on temporary tables are performed. Care is also taken, so that no deadlock problems are introduced, as locking is done at session level.

Future work includes support for multiple interval granularities, multiple valid time dimensions, transaction support as well as multi-user extensions.

7. References.

[1] C. J. Date, “An Introduction to Database Systems”, Vol. II, Addison-Wesley Publishing Company.

[2] A. S. Tanenbaum, “Modern Operating Systems”, Prentice Hall Inc., 1992.

[3] C. Vassilakis, N. Lorentzos, P. Georgiadis and Y. Mitsopoulos, “ORES: Design and Implementation of a Temporal DBMS”, Submitted for publication.

[4] N. A. Lorentzos, “The interval extended relational model and its application to valid time databases, Temporal Databases: Theory, Design and Implementation”, J. Clifford, R. Snodgrass et al (Ed.), Benjamin Cummings, pp. 67-91, 1993.

[5] ESPRIT III Project 7224 (ORES) Deliverable D2: “Specification of Valid Time SQL”, April 1993.

[6] ORACLE Corporation, “SQL Language Reference Manual” (for version 6.0), 1990.

[7] Sybase Inc., “Transact SQL User’s Guide” (for release 4.2), 1990.

[8] E. F. Codd, “A relational model of data for large shared data banks”, Communications of the ACM, 13(6), 377-387, June 1970.

[9] ESPRIT III Project 7224 (ORES) Deliverable C3: “Specification of Valid Time Formalism”, April 1993.

[10] C. Vassilakis, P. Georgiadis and N. Lorentzos, “An Optimised Implementation of a Temporal DBMS”, Submitted for publication.

[11] Ingres Corporation, “Ingres SQL and ESQL Reference Manual” (for release 6.4), 1991.