

# IMPLEMENTATION OF TRANSACTION AND CONCURRENCY CONTROL SUPPORT IN A TEMPORAL DBMS

COSTAS VASSILAKIS<sup>1</sup>, NIKOS LORENTZOS<sup>2</sup>, PANAGIOTIS GEORGIADIS<sup>1</sup>

<sup>1</sup> Department of Informatics, University of Athens, TYPA Builds., 157 71, Zografou, Athens, Greece

<sup>2</sup> Informatics Laboratory, Agricultural University of Athens, Iera Odos 75, 118 55, Athens, Greece

**Abstract** — Transactions and concurrency control are significant features in database systems, facilitating functions both at user and system level. However, the support of these features in a temporal DBMS has not yet received adequate research attention. In this paper, we describe the techniques developed in order to support transaction and concurrency control in a temporal DBMS that was implemented as an additional layer to a commercial DBMS. The proposed techniques make direct use of the transaction mechanisms of the DBMS. In addition, they overcome a number of limitations such as automatic commit points, lock release and log size increment, which are imposed by the underlying DBMS. Our measurements have shown that the overhead introduced by these techniques is negligible, less than 1% in all cases. The approach undertaken is of general interest, it can also be applied to non-temporal DBMS extensions.

*Key words:* Data Modelling, Temporal Databases, Transactions, Concurrency control.

## 1. INTRODUCTION

Transactions and support of concurrency control represent features of a Database Management System (DBMS) which are of major practical importance. Concurrency control support is significant because it increases the degree of parallelism and system throughput, protecting against operation interference which can lead to inconsistencies in the database and/or production of erroneous results [1]. Transactions at system level constitute the unit of *sharing* and *recovery* [2] since locks acquired during a DBMS session are released at the end of the current transaction and, when a system failure occurs, the database is restored to a previous state, at a *COMMIT point*. At user level, transactions provide an *atomic operation abstraction* [3], which eases programming tasks. They constitute the unit of *integrity* [2], by allowing the transition from one valid database state to another, via an invalid state, and they facilitate *undoing* erroneous changes.

Today, commercial DBMSs do support transactions and concurrency control. However, the literature reveals many prototype implementations, aiming at extending the functionality of such a DBMS, to areas of special interest. Such implementations often include new relational algebra operations, usually developed on *top* of the DBMS. As a consequence, these extensions cannot take advantage of the transaction and concurrency control support, provided by the DBMS, for the following reasons:

1. In some cases the new algebraic operations create temporary tables, to store temporary results. In some DBMSs, however, issuing a DDL statement like CREATE TABLE results in an *implicit commit point* [4]. This deprives the ability to use the ROLLBACK statement and *undo* changes up to the database state before the implicit commit point. In some other DBMSs, the usage of DDL statements in multi-statement transactions is disallowed [5].
2. Writing results to temporary tables constitutes a change to the database state, thus writing is logged. Since the algorithms used to execute an enhanced operation may produce a considerable amount of temporary data, the requirements for log space may increase dramatically. Thus, techniques have to be developed, to reduce log space requirements.
3. Locks acquired during a session should persist along commit points introduced either implicitly by the DBMS or explicitly by the enhanced operations, in order to (i) protect from interference between operations and (ii) guarantee that only legal states of the database are visible.

The above observations also apply to temporal extensions to the relational model, an area that has attracted the interest of many researchers in recent years: Two international workshops have been organised (Texas, 14-16 June 1993; Zurich, 17-18 Sept. 1995), one international seminar (Dagstuhl, 23-27 June, 1997) and a book has been published [6], all of them dedicated to this area. It can be noticed, however, that although many temporal models have been proposed for the management of temporal data ([7-13], are only some representative approaches), few of them have been implemented [14]. Moreover, to our knowledge, nothing has been reported on the transaction and concurrency control support in a temporal DBMS.

This paper aims at filling in this gap, by providing a description of the algorithms implemented in an actual development of a layered temporal DBMS. The language of the temporal DBMS is VT-SQL [15], a consistent *Valid Time Extension* to SQL89. The specification of VT-SQL has been based on the IXRM model [16]. A further extension of VT-SQL can be found in [17]. Details on the design and implementation of the temporal DBMS and VT-SQL can be found in [18].

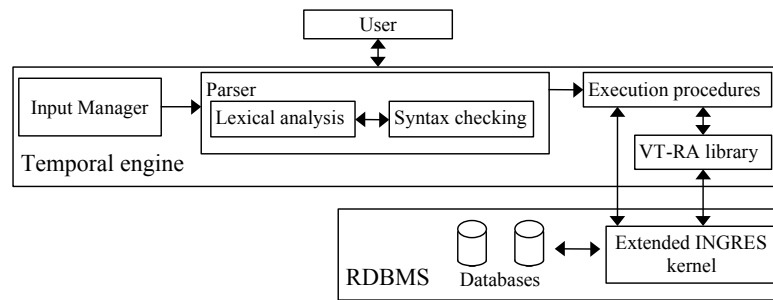


Fig. Error! Unknown switch argument.. Architecture of the temporal DBMS.

The overall architecture of the temporal DBMS can be seen in Figure 1. The temporal DBMS consists of two layers: The upper layer is the *Temporal Engine*. This layer contains, amongst others, the VT-RA library, which implements the operations of *Valid Time Relational Algebra* [16, 19]. It can easily be ported to other DBMSs, since it has been coded in C and embedded SQL and thus it is independent of the internal DBMS architecture. The lower layer is the INGRES DBMS, whose kernel has been extended to support one additional data type, *DATEINTERVAL*, along with predicates and functions. The Temporal Engine and the Extended INGRES kernel have been implemented within the ORES project. The algorithms for transaction and concurrency control support which are described in the present paper were designed and implemented in a follow-up research work. These algorithms concern an implementation based on INGRES. However, the following should be noted:

1. Wherever necessary, it is explained how equivalent results can be obtained in implementations based on commercial DBMSs other than INGRES.
2. The algorithms are of general interest, since they can be adapted appropriately, to commercial DBMS extensions other than temporal.
3. The algorithms do not introduce additional mechanisms for transaction and concurrency control support, but exploit the ones offered by the underlying DBMS, thus minimising the introduced overheads.

The remainder of the paper is organised as follows: In section 2, VT-RA is presented in brief. Section 3 outlines the syntax and semantics of VT-SQL. Section 4 presents the algorithms developed for the execution of the VT-SQL DML statements which provide transaction and concurrency control support. Graphs are also plotted, which allow an estimation of the time overhead introduced by the presented techniques. Conclusions and future work are outlined in the last section.

## 2. VALID TIME RELATIONAL ALGEBRA

The theoretical foundation of the temporal DBMS is Valid Time Relational Algebra (VT-RA) [16], a consistent extension to Codd's algebra [20]. Time is represented using two new data types, namely

DATE and DATEINTERVAL. The YYYY-MM-DD format is used for date literals, which is a more readable form of the ANSI standard YYYYMMDD. The notation  $[d_i, d_j)$  is used to represent values of type DATEINTERVAL, where  $d_i$  and  $d_j$  are dates and  $d_j$  is greater than  $d_i$ . The first of these dates ( $d_i$ ) is called the *start* whereas the second ( $d_j$ ) is called the *stop* of  $[d_i, d_j)$ . A DATEINTERVAL value contains all the dates from  $d_i$  and up to, but not including  $d_j$ . VT-RA defines predicates for interval comparison, transformations between the representations of time (points and intervals) and temporally extended versions of operations UNION and EXCEPT, which can be applied to relations containing attributes of type DATE and/or DATEINTERVAL. These operations are described briefly in the subsections that follow. For a more detailed presentation, see [16, 19].

### 2.1. Operation Fold

Let R be a relation whose schema is  $(A_1, A_2, \dots, A_n)$  and assume that the domain of attribute  $A_i$  is either DATE or DATEINTERVAL. When R is *folded* on column  $A_i$  (denoted as FOLD $[A_i]$  (R)), all its tuples (i) whose  $A_j$  columns have identical values  $\forall j \neq i$ , and (ii) their  $A_i$  columns can *merge* (i.e. they can form a single DATEINTERVAL) are replaced in the resulting relation by a single tuple, with the same values in all  $A_j$  columns,  $\forall j \neq i$ , but the value of the  $A_i$  column is formed by the *merging* of the  $A_i$  column of these tuples. For example, if ASSIGNMENT is any of the relations in Figure 2, then FOLD[Time] (ASSIGNMENT) yields the relation in Figure 3.

Operation FOLD may be applied to a relation R on multiple columns  $A_{i_1}, A_{i_2}, \dots, A_{i_n}$  of a DATE or DATEINTERVAL type. This is denoted as FOLD  $[A_{i_1}, A_{i_2}, \dots, A_{i_n}]$  (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  $A_{i_n}$ .

ASSIGNMENT		
Name	Department	Time
Mary	Toys	d1
Mary	Toys	d2
...	...	...
Mary	Toys	d4
Mary	Toys	d10
...	...	...
Mary	Toys	d14
John	Sales	d1
...	...	...
John	Sales	d19

(a)

ASSIGNMENT		
Name	Department	Time
Mary	Toys	[ d1, d3)
Mary	Toys	[ d2, d5)
Mary	Toys	[d10, d15)
John	Sales	[d1, d10)
John	Sales	[d10, d15)
John	Sales	[d15, d18)
John	Sales	[d16, d20)

(b)

Fig. 2. Two valid time relations.

ASSIGNMENT		
Name	Department	Time
Mary	Toys	[ d1, d5)
Mary	Toys	[d10, d15)
John	Sales	[ d1, d20)

Fig. 3. A valid time relation.

### 2.2. Operation Unfold

Let R be a relation whose schema is  $(A_1, A_2, \dots, A_n)$  and assume that the domain of some attribute  $A_i$  is DATE or DATEINTERVAL. Notice that if  $A_i$  is of type DATE and  $t_j$  is a value recorded in  $A_i$ , then  $t_j$  can be seen as a *trivial* interval  $[t_j, t_{j+1})$  that contains exactly one date,  $t_j$ . Hence, when R is unfolded on attribute  $A_i$  (denoted as UNFOLD  $[A_i]$  (R)), each tuple  $(t_1, \dots, t_{i-1}, t_i, t_{i+1}, \dots, t_n)$  of R is replaced in the resulting relation by a family of tuples  $(t_1, \dots, t_{i-1}, t_{ij}, t_{i+1}, \dots, t_n)$ , where each  $t_{ij}$  is a date included in  $t_i$ . For example, if ASSIGNMENT is the relation in Figure 3, then UNFOLD[Time] (ASSIGNMENT) yields the relation in Figure 2(a). An UNFOLD may apply to multiple columns of either a DATE or

DATEINTERVAL type; this is denoted as  $\text{UNFOLD}[A_{i_1}, A_{i_2}, \dots, A_{i_n}](R)$  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. Operation Normalise

Operation NORMALISE is a synthesis of the FOLD and UNFOLD operations and may be applied on multiple columns  $A_{i_1}, A_{i_2}, \dots, A_{i_n}$  of type DATE or DATEINTERVAL. It is denoted as  $\text{NORMALISE}[A_{i_1}, \dots, A_{i_n}](R)$  and is *semantically* equivalent to  $\text{FOLD}[A_{i_1}, \dots, A_{i_n}](\text{UNFOLD}[A_{i_1}, \dots, A_{i_n}](R))$ .

### 2.4. Operation PUnion

The PUNION (*point*-union) 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 as  $R \text{ PUNION}[A_{i_1}, \dots, A_{i_n}] S$ , where  $A_{i_1}, A_{i_2}, \dots, A_{i_n}$  are of a DATE or DATEINTERVAL type. This operation is *semantically* equivalent to  $\text{NORMALISE}[A_{i_1}, \dots, A_{i_n}](R \text{ UNION } S)$

For example, if a relation  $S$  has a single tuple,

(Mary, Toys, [d3, d12))

and ASSIGNMENT is the table in Figure 3, then the result of ASSIGNMENT PUNION [Time]  $S$  consists of the two tuples

(Mary, Toys, [d1, d15))  
(John, Sales, [d1, d20))

### 2.5. Operation PExcept

Operation PEXCEPT (*point*-except) can be applied to two union-compatible relations and operates on multiple columns of type DATE or DATEINTERVAL. The application of PEXCEPT on two relations  $R$  and  $S$  is denoted as  $R \text{ PEXCEPT}[A_{i_1}, \dots, A_{i_n}] S$  where  $A_{i_1}, A_{i_2}, \dots, A_{i_n}$  are of type DATE or DATEINTERVAL. This operation is *semantically* equivalent to

$\text{FOLD}[A_{i_1}, \dots, A_{i_n}](\text{UNFOLD}[A_{i_1}, \dots, A_{i_n}](R) \text{ EXCEPT UNFOLD}[A_{i_1}, \dots, A_{i_n}](S))$ .

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

(Mary, Toys, [d10, d15))  
(John, Sales, [d10, d15))

and ASSIGNMENT is the table in Figure 3, then ASSIGNMENT PEXCEPT [Time]  $S$  consists of the tuples

(Mary, Toys, [ d1, d5))  
(John, Sales, [ d1, d10))  
(John, Sales, [d15, d20))

Tables for which data insertion and deletion is performed via PUNION and PEXCEPT rather than via UNION and EXCEPT, respectively, are called *normalised*.

## 3. THE VT-SQL LANGUAGE

In general, VT-SQL does not extend the DDL statements of SQL. One exception is the CREATE TABLE statement whose extension (see section 3.2, below) accommodates new clauses associated to the semantics of valid time tables and the specification of the primary key of such a table. Hence, the syntax and semantics of the VT-SQL DML statements are presented in the subsections that follow. For a complete presentation of the VT-SQL syntax and semantics, the user is referred to [15]. The following conventions are used for the syntax presented hereinafter: Terms enclosed in square brackets ([ ]) are

optional. Braces ({} ) are used for items that may be repeated zero or more times. Parentheses are used for grouping. Single quotes are used for parentheses that must be typed literally. Capitals indicate reserved words. Finally, italics denote user-provided values.

### 3.1. Statement Select

The syntax for the VT-SQL 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) ResultColumnList}]
[NORMALISE ON ResultColumnList]
```

(Note that in [15] another version of UNFOLD, namely UNFOLD ALL, is also described but has been omitted here, for brevity reasons.) An *extended-select* is executed by evaluating its *sql-select* part and then applying the REFORMAT and/or NORMALISE operations stated in 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 evaluated. Assuming that the schemata of the results of the two *extended-selects* are union-compatible, the relevant VT-RA infix operation (some version of UNION or EXCEPT) is applied to them in order to evaluate the final query outcome. From these operations, UNION 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 PUNION 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. However, UNION ALL ResultColumnList returns the same result with UNION ResultColumnList. Analogously, the EXCEPT keyword specifies that either PEXCEPT or EXCEPT 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. Statement Insert

The syntax of the VT-SQL INSERT statement is identical with its SQL counterpart, except when the tuples to be inserted are specified by a query. In this case the query may be an *extended-select*, i.e. it may include the REFORMAT AS and/or NORMALISE clause. When data are inserted into 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 (see definition at end of Subsection 2.5), then the insertion of the specified data is followed by a NORMALISE operation on the appropriate columns.

A normalised table may optionally have a key. For example, the key of ASSIGNMENT, in Figure 3, is <Name, Time-*p*>, which means that table ASSIGNMENT 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 DATEINTERVAL values which have at least one date in common. We say that within table ASSIGNMENT primary key uniqueness is preserved *at a date level*.

If a normalised relation R has a primary key and an attempt is made to insert into it a piece of data that has already been recorded in R, then the insertion fails. For example consider ASSIGNMENT, in Figure 3, with key <Name, Time-*p*>. If we issue the command

```
INSERT INTO ASSIGNMENT
VALUES ('Mary', Toys, '[d4, d10]')
```

then the insertion will fail, because Mary's assignment for date *d4* is already recorded in ASSIGNMENT. This convention is a consistent extension to standard SQL.

### 3.3. Statement Delete

In VT-SQL the syntax of statement DELETE has been extended and includes an optional PORTION clause, which may be used when deleting data only from a normalised table. If this clause is missing the deletion applies in the known way. If the DELETE statement contains the PORTION clause then the deletion applies to the valid time period specified by this clause. For example, after the execution of the command

```
DELETE FROM ASSIGNMENT
PORTION Time = '[d3, d12)'
WHERE Name = 'Mary'
```

table ASSIGNMENT, in Figure 3, will consist of the tuples

```
(Mary, Toys, [ d1, d3))
(Mary, Toys, [d12, d15))
(John, Sales, [ d1, d20))
```

### 3.4. Statement Update

If an update statement is issued against a non-normalised table, then the standard SQL semantics apply to the update operation. If, however, the statement is issued against a normalised table, then data modification is always followed by a NORMALISE operation on the appropriate columns. Analogously to the DELETE statement, when the UPDATE statement is applied to a normalised table it may optionally contain a PORTION clause, which designates the valid time period during which the update is applicable. For example, the command

```
UPDATE ASSIGNMENT
PORTION Time = '[d3, d5)'
SET Name = 'Tom'
WHERE Name = 'Mary'
```

will result in that ASSIGNMENT, in Figure 3, will consist of the tuples

```
(Mary, Toys, [ d1, d3))
(Mary, Toys, [d10, d15))
( Tom, Toys, [ d3, d5))
( John, Sales, [ d1, d20))
```

If primary key uniqueness of the updated table is preserved *at a date level*, the update is again rejected whenever this uniqueness is violated.

## 4. TRANSACTION AND CONCURRENCY CONTROL SUPPORT

In a layered temporal DBMS, transaction and concurrency control support can be implemented by using two *sessions* between the Temporal Engine and the DBMS. The first session is the *system session*, which is used to create and modify the temporary tables. The second session is the *user session*, which is used to access and modify the user tables. The user session may access the temporary tables for reading only.

Since temporary tables are created and modified through the system session, no implicit commit points are introduced for the user session. Also, the increment of the user session's log size is kept low. Finally, the acquired locks will not be released until the current transaction of the user session is explicitly committed or rolled back. However, since different sessions to the DBMS are competing for locks on the data, care must be taken so that the access scheme through the two sessions does not lead to deadlocks.

The usage of the two sessions that provide transaction and concurrency control support is described in the subsections that follow. Lock acquisition on behalf of the Temporal Engine is analysed and it is shown that the temporal DBMS guarantees serialisability of operations, if the underlying DBMS provides serialisable transactions. For a discussion, on how the following algorithms are used for transaction support as well as for a protection and recovery scheme for temporary tables, see [21].

The algorithms presented in this section may be implemented on top of any DBMS which provides savepoints and elementary lock control statements. They have been implemented on a Sun SPARC ELC workstation, running SunOS 4.1.3, and INGRES 6.4. They have proved to be efficient. Compared in particular with an implementation which uses a single session to the underlying DBMS (and thus does not support transactions or concurrency control), the performance of the algorithms is only up to 0.95% worse than their single-session counterparts. The subsections for statements *INSERT*, *DELETE* and *UPDATE* include performance figures that indicate performance measures of the respective algorithms.

#### 4.1. Statement Select

Three cases are considered for the evaluation of the *SELECT* statement (for a complete description, see [15]):

**Case (i):** *The user's query includes only algebraic operations supported directly by the RDBMS.*

In this case the *SELECT* statement matches that of standard SQL, except that it may also contain *DATE* and/or *DATEINTERVAL* predicates and functions. Then the Temporal Engine opens a cursor through the user session, which retrieves the result of the user query. The result tuples are fetched through this cursor and presented to the user.

**Case (ii):** *The user's query is a single extended-select.*

In this case the query has the form *SELECT-FROM-WHERE- GROUP BY-HAVING* followed by a *REFORMAT AS* and/or a *NORMALISE ON* clause. Then the following steps are taken:

- A. The system session is used to create a temporary table whose schema matches the schema of the table resulting from the *sql-select* part of the *extended-select*. A cursor is opened through the user session, retrieving the result of the *sql-select* part of the user query.
- B. The result tuples of the *sql-select* are fetched through the cursor opened in step (A) and are subsequently 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 system session is used to execute the operations specified by the *REFORMAT* and/or *NORMALISE* clauses. As soon as each operation stated in these clauses completes, the temporary table holding the result of the previous step is dropped through the system session and the system session commits.
- D. The tuples contained in the final temporary table are fetched through the system session and forwarded to the user. When all data have been exhausted, the final temporary table is dropped through the system session and the system session commits.

**Case (iii):** *The user's query consists of two extended-select statements.*

In this case the two statements are combined by some version of either *UNION* or *EXCEPT*. Then the following procedure is used:

- A. Steps (A)-(C) of Case (ii) above are performed to evaluate each of the *extended-select* statements, storing the results in temporary tables. (If the *extended-select* does not contain the *REFORMAT* and *NORMALISE* clauses, then only steps (A) and (B) are performed).
- B. The system session is used to apply the appropriate binary operation (some version of either *UNION* or *EXCEPT*) to the temporary tables produced in step (A), and the result is stored in a temporary table. Upon operation completion, the temporary 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.

Data in user tables, which contain the final query outcome, are accessed through the user session in the procedure described above. Thus the user session will acquire shared locks on that data, making them unavailable for modification through other sessions, while the current transaction of the user session is active (i.e. has not been committed or rolled back).

#### 4.2. Statement Insert

In order to execute an INSERT statement, the normalisation status of the target table is checked and, depending on whether the target table is normalised or not, the appropriate algorithm is selected, as described in the next two subsections. In the following, the table into which data will be inserted is denoted as  $R$ . The columns of any valid time table  $T$  are denoted as  $T_{c1}, T_{c2}, \dots, T_{cN}, T_{vt}$ , with  $T_{vt}$  being the column storing the valid time of the tuple. All the columns from those in  $T_{c1}$  through  $T_{cN}$  that do not participate in the primary key are denoted as  $T_{-key}$ . If no primary key is defined on  $T$ , then  $T_{-key}$  includes all the columns from  $T_{c1}$  through  $T_{cN}$ . Except for  $T_{vt}$ , all other columns participating in the primary key are referenced as  $T_{key}$ .

##### 4.2.1. The target table is not normalised

Two sub-cases are considered, depending on how the values to be inserted are specified:

**Case (i):** The values to be inserted are specified either by means of the *VALUES* clause or by a *SELECT* statement that does not contain the *REFORMAT* and *NORMALISE* clauses.

In this case the DBMS can handle the data insertion on its own, so the Temporal Engine forwards the INSERT statement to the DBMS through the user session. The DBMS executes the command, imposing the appropriate locks on behalf of the user session (i.e. exclusive locks on the inserted data; if the inserted values are specified by means of a *SELECT* statement shared locks on the used data are also imposed).

**Case (ii):** The values to be inserted are specified by means of an extended *SELECT* statement containing the *REFORMAT* and/or *NORMALISE* clauses.

In this case the extended-select statement is evaluated, following the steps (A)-(C) of Case (ii) of the *SELECT* statement execution algorithm, and the resulting tuples are stored into a temporary table denoted here as  $T1$ . (The name of the table is actually a unique string, computed by the Temporal Engine. The string contains the system session's *session identifier* and the current timestamp, ensuring in this way that different sessions to the DBMS will use different temporary table names.) Subsequently, the command

```
INSERT INTO R SELECT * FROM T1
```

is forwarded to the DBMS through the user session. Finally, the system session is used to drop table  $T1$  and the system session commits.

Since the INSERT command which is forwarded to the DBMS through the user session accesses table  $T1$ , which is subsequently dropped through the system session, it is important that this access does not result in lock imposition on the accessed data because a deadlock will then occur. In order to achieve this, the command

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

is forwarded to the DBMS, through the user session, prior to the INSERT command. (The *SET LOCKMODE* statement is an INGRES extension to SQL89 ([22]); the syntax in other DBMSs is different.)

In the algorithm presented here, the user session will acquire shared locks on the used data during the query evaluation phase. Additionally, during the insertion phase the same session will obtain exclusive locks on the inserted tuples making them unavailable for update and access, while the current transaction of the user session is active.

##### 4.2.2. The target table is normalised

In this case four distinct sub-cases are considered, depending on (i) whether a primary key is defined for the table and (ii) how the values to be inserted are specified:

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

The Temporal Engine creates an *insertion tuple*, holding the values specified in the *VALUES* clause. It also opens a cursor *ins\_cur* on table  $R$  through the user session, selecting all the tuples which are *value equivalent* to the insertion tuple (i.e. each column in  $R_{-key}$  is equal to the corresponding column in the insertion tuple) and have overlapping or adjacent valid times. Each selected tuple is deleted from table  $R$



using cursor *ins\_cur* (for which the selected tuple is current) and the valid time of the insertion tuple is replaced by the union of its former value and the valid time of the deleted tuple. (The union of two adjacent or overlapping dateintervals is a dateinterval containing all the time points in both arguments). Finally, the insertion tuple is appended to *R* through the user session.

In the procedure described above, the user session obtains exclusive locks for both the deleted tuples and the inserted tuple. Thus these pieces of data are unavailable to other sessions, while the current transaction of the user session is active.

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

As in the previous case, the Temporal Engine creates an insertion tuple, holding the values specified in the VALUES clause. A *savepoint InsSave* is created for the user session and a cursor *ins\_cur* is opened on *R* through the user session, selecting all the tuples for which: (i) All the columns in  $R_{key}$  have values equal to the values of the corresponding table in the insertion tuple. (ii) Their valid times are adjacent or overlapping to the timestamp of the insertion tuple. For each qualifying tuple, the values for  $R_{key}$  and  $R_{vt}$  are fetched into the Temporal Engine's memory, and 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 *InsSave*, by issuing a  
`ROLLBACK TO InsSave`  
statement through the user session. Further processing is aborted.
2. If all columns in  $R_{key}$  of the fetched tuple have values equal to the corresponding columns of the insertion tuple, then: Firstly, the fetched tuple is deleted from *R* through cursor *ins\_cur*. Secondly, 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  $R_{key}$  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.

In the above algorithm, the user session acquires exclusive locks both on the deleted tuples and the inserted tuple, making them all unavailable to other sessions, while the current transaction of the user session is active. However, shared locks will be placed on the tuples which will be selected by cursor *ins\_cur* but cannot be coalesced with the insertion tuple, in the case that the value of some column of  $R_{key}$  is different in the insertion tuple and the selected tuple. These extra locks are not a major problem for the following reasons:

1. Considering that a primary key has been defined on table *R*, the number of tuples that will unnecessarily be locked is restricted to at most two (one tuple whose valid time ends at the start of the insertion tuple's valid time, and another whose valid time starts at the end of the insertion tuple's valid time).
2. If these two tuples do exist, they will probably be located in the disk page where the insertion tuple will be inserted. The reason is that the storage structures usually used for tables on which primary keys are defined, partition tuples with respect to their primary key values (e.g. B-tree or ISAM on the columns participating in the primary key). Considering that DBMSs usually place locks at *disk page* level rather than at tuple level, the exclusive lock on the inserted tuple would affect these tuples, in any case.

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

The *extended-select* is evaluated as described in Subsection 4.1 and the result is stored in a temporary table, by following steps (A)-(C) of Case (ii) of the SELECT statement execution algorithm. (If the *extended-select* does not contain the REFORMAT and NORMALISE clauses, then only steps (A) and (B) are performed). In the sequel, this temporary table is denoted as *T1*. 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  $R_{key}$  has value equal to the corresponding column in  $T1_{key}$  and the value of  $R_{vt}$  is overlapping or adjacent to the value of  $T1_{vt}$ ). Since evaluating the query associated with this cursor implies access to table *T1*, which will be dropped afterwards through

the system session, it is important that this access does result in lock placing 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. Each qualifying tuple is fetched into memory and deleted from  $R$  through the user session and subsequently inserted through the system session into table  $T1$ . Afterwards, the system session is used to execute FOLD [ $T1_{vt}$ ] ( $T1$ ), each tuple of the FOLD operation's result is fetched into memory through the system session and inserted into  $R$  through the user session. Finally, the temporary tables are dropped through the system session and the system session commits.

In the procedure described above, the user session acquires shared locks on the tuples which are used to compute the query result during the evaluation of the (*extended-*) *select* and during the insertion phase. The user session obtains exclusive locks on the tuples that are deleted from table  $R$  or inserted into it; thus these tuples are not accessible by other sessions, while the current transaction of the user session is active.

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

A *savepoint* is introduced for the user session, the *extended-select* is evaluated and the result is stored in a temporary table  $T1$ , as in the previous case. Afterwards, the statement

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

is forwarded to the DBMS through the user session, specifying that accesses to table  $T1$  should not place locks; the same session is used to open a cursor on the join of  $R$  with  $T1$ , enabling to select subsequently from  $R$  those tuples for which (i) all columns in  $R_{key}$  have values equal to the corresponding columns of some tuple in table  $T1$  and (ii)  $R_{vt}$  is either overlapping or adjacent to the valid time of the same tuple of table  $T1$ . For each one of these tuples, the values on all the columns of table  $R$  and on all the columns of  $T1_{key}$  and  $T1_{vt}$  are fetched into main memory. Next, the following checks are made:

1. If the values of  $R_{vt}$  and  $T1_{vt}$  overlap then 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, and further processing is aborted. Table  $T1$  is dropped and the system session commits.
2. If the values of  $R_{vt}$  and  $T1_{vt}$  are adjacent and all columns in  $R_{key}$  have values equal to the corresponding columns in  $T1_{key}$ , then the current tuple of  $R$  is deleted through the user session and it is subsequently inserted into  $T1$  through the system session. (Note that the values are currently into the main memory, so they can be inserted immediately).
3. In all other cases, i.e. if the values of  $R_{vt}$  and  $T1_{vt}$  are adjacent and some column in  $R_{key}$  is not equal to the corresponding column of  $T1_{key}$ , 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  $T1_{key}$ ,  $T1_{vt}$ ,  $T1_{key}$ , 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 have been exhausted, *working tuple* is inserted into  $R$  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 either the value of any of the columns in  $T1_{key}$  is different in *working tuple* and *current tuple* or the values for  $T1_{vt}$  in the two tuples are neither overlapping nor adjacent, then *working tuple* is inserted into  $R$  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 through the user session introduced at the beginning of the algorithm and further processing is aborted. Table  $T1$  is dropped and the system session commits.
4. If the value of any of the columns in  $T1_{key}$  is different in *working tuple* and *current tuple*, then *working tuple* is inserted into  $R$  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*, *current tuple* is discarded, and control passes to step (1).
5.  $T1$  is dropped through the system session and the system session commits.

Analogously to Case (ii), shared locks may be unnecessarily placed on some tuples of table  $R$  (up to two tuples may be locked for each tuple in table  $T1$ ), but these extra locks do not reduce data availability or operation concurrency for the reasons already stated in Case (ii)).

One point that has to be taken into account for Case (ii) of Subsection 4.2.1 and Cases (iii) and (iv) of Subsection 4.2.2 is that some DBMSs (e.g. INGRES) allow the usage of locking control statements, such as *SET LOCKMODE*, only when they appear at the beginning of multi-statement transactions. If the Temporal Engine operates on top of such a DBMS, then the default read locking mode of the user session should be set to 'no locking', when the session is created. Also, the user should use the *SET LOCKMODE* command, to specify the tables to which read access should result in locking. Note that no locking capabilities are lost because of this modification; only the *default locking scheme* of the DBMS is altered.

#### 4.2.3. Performance Evaluation

The performance of the insertion algorithm presented above is evaluated in Figures 4 and 5 for insertions into a table *SALARY*(Name, Amount, Period). Each diagram depicts the execution time overhead, introduced by the dual session algorithm, as a percentage of the overall execution time. As can be seen in the diagrams, measurements were obtained for different numbers of affected tuples and various database sizes. The specifications of the tuples in table *SALARY* are as follows: Groups of 10-50 tuples share the same name. For each name, the salary values are distributed uniformly. The bounding dates for period are drawn from the domain [1980-01-01, 2029-01-01). For each name, the start of the period and the duration of a period are distributed uniformly. The storage structure of *SALARY* is *heap* (i.e. unstructured and unkeyed), and an index has been defined on all the table columns. All the measurements were obtained on a Sun SPARC ELC workstation with 24 Mbytes of memory and 1 GByte disks, running SunOS 4.1.3, and INGRES 6.4, where the implementation took place.

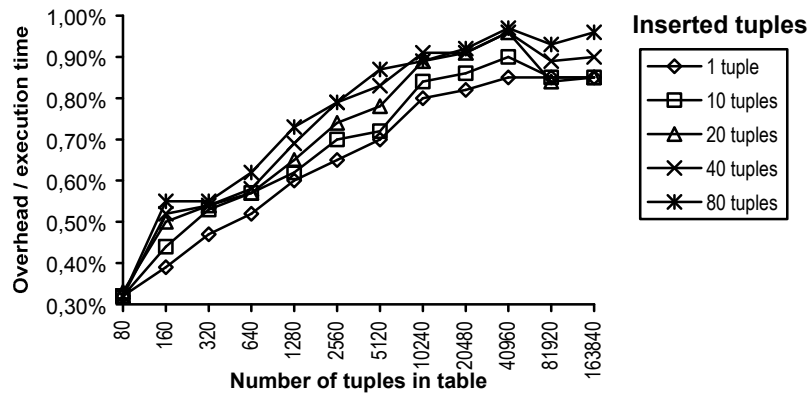


Fig. 4. Insertion into a table with no primary key.

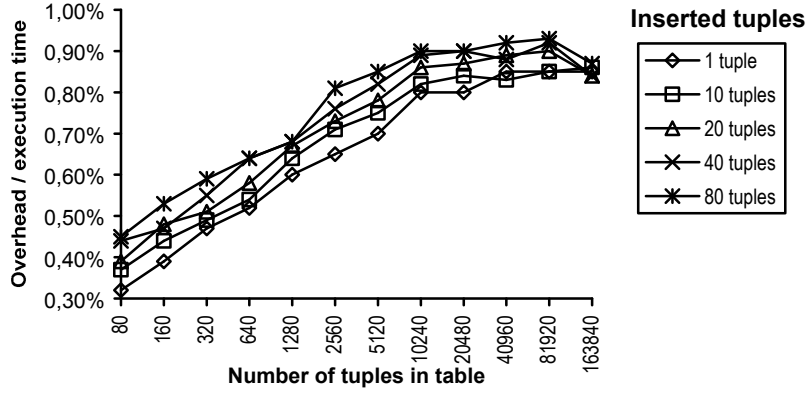


Fig. 5. Insertion into a table with a primary key.

As can clearly be seen by the diagrams, the overhead introduced by the use of the dual session technique is negligible. Our measurements showed that in all the cases that follow, this overhead was less than 1%. This has been achieved because the dual-session algorithms do not introduce any extraneous I/O, compared to the single-session algorithms and, at the same time, they take advantage of the transaction and concurrency control algorithms of the underlying DBMS. The only overheads introduced by the dual-session algorithms are:

1. The administrative overhead of maintaining two sessions and switching between them.
2. The overhead for managing the savepoints; this should be minimal, since savepoints are typically implemented by writing a record into the transaction's log.
3. The overhead due to the retrieval of data via some session and its storing from within another session (e.g. the first step in Case (ii) of Subsection 4.2.1 and Cases (iii) and (iv) of Subsection 4.2.2). One example is when a temporary table is filled in with data from user tables in which case data are moved from the DBMS's address space to the Temporal Engine's address space. As opposed to this, in the single-session algorithms the temporary table is filled in using a bulk insertion statement (`INSERT INTO T1 SELECT . . .`); this statement fills in the temporary table without any intervention of the Temporal Engine. It should be noted however that even when the number of these tuples is high, this overhead is low, since it accounts only for interprocess communication and process switching costs, i.e. operations that are *cheap* compared with I/O.

#### 4.3. Statement Delete

If the `PORTION` clause is not specified in the `DELETE` statement (which is always the case for non-normalised tables, as the usage of this clause is restricted to normalised tables), the request is directly forwarded to the underlying DBMS for execution through the user session.

If the `PORTION` clause is present, the Temporal Engine opens, through the user session, a cursor on the target table  $R$ , selecting the rows which satisfy the `WHERE` clause and have valid times overlapping with the valid time (denoted as *period*, hereinafter) specified in the `PORTION` clause. For each qualifying 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  $R$  through the user session.
2. If the difference  $R_{vt} - \text{period}$  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 updated to  $R_{vt} - \text{period}$ , through the user session. This is achieved by the use of the `WHERE CURRENT OF` form of the embedded SQL `UPDATE` statement.
3. If the difference  $R_{vt} - \text{period}$  yields two intervals, *diff1* and *diff2*, the value of  $R_{vt}$  of the current tuple is updated to *diff1*. Also, a new tuple is appended to  $R$  whose values for  $R_{cl}$ , ...,  $R_{cN}$  are equal to the

values of the corresponding columns in the current tuple, whereas the value of column  $R_{vt}$  is set equal to  $diff2$ . Both the update and the tuple insertion are performed through the user session.

Following the algorithm presented above, the user session will acquire exclusive locks on the inserted, deleted and updated tuples; these tuples will be inaccessible to other sessions, while the current transactions of the user session is active. If the DELETE statement contains a WHERE clause which includes subqueries referencing other tables, the user session will also obtain shared locks on the data used to evaluate the WHERE clause, as well.

#### 4.3.1. Performance Evaluation

The performance of the deletion algorithm presented above is evaluated in Figure 6. Each diagram depicts the execution time overhead, introduced by the dual session algorithm, as a percentage of the overall execution time. Again, measurements were obtained for different numbers of affected tuples and various database sizes. Only the case of deletion using the *PORTION* clause was considered, since deletion without using the *PORTION* clause is handled by the DBMS. The details about the data in the base table, the storage structure, and the implementation platform are those described in Subsection 4.2.3. Since the deletion algorithm does not use the second session, no differences in performance are actually observed.

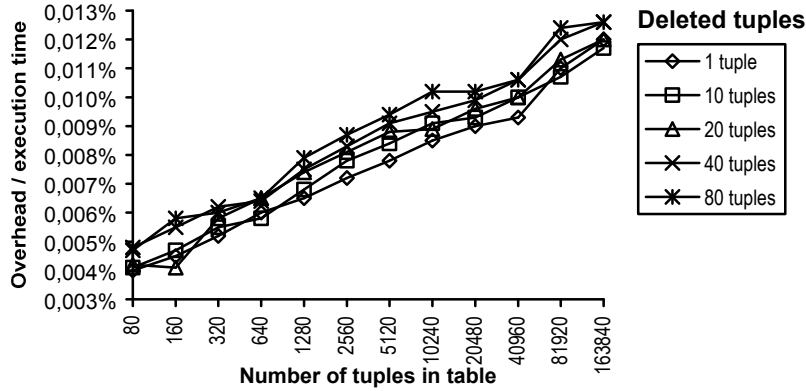


Fig. 6. Deletion, using the PORTION clause.

#### 4.4. Statement Update

If the UPDATE statement is applied to a non-normalised table, it is directly forwarded to the underlying DBMS for execution. If, however, the target table  $R$  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 retrieves through the user session all the tuples which satisfy the WHERE clause. For each selected tuple, the updated values of the fields changed by the SET clause are fetched rather than the original values. The selected tuple is deleted from the table through the user session and a tuple containing the updated values is stored through the system session in a temporary table, *update\_temp*. (This table is created through the system session). When all qualifying tuples have been fetched, the system session commits, and the algorithm described for Case (iii) of Subsection 4.2.2 is employed to insert the tuples stored in *update\_temp* into  $R$ . (The step involving the execution of the *extended-select* is not performed. Table  $T1$  mentioned in Subsection 4.2.2 is actually the *update\_temp* table, produced in the previous step).

In the procedure presented above, the user session will acquire the necessary exclusive locks on the affected tuples, making them unavailable to other sessions, while the current transaction of the user session is active. If the UPDATE statement contains a WHERE clause which includes subqueries

referencing other tables, the user session will also obtain shared locks on the data used to evaluate the WHERE clause.

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

The Temporal Engine opens a cursor on  $R$  through the user session, selecting the tuples which (i) satisfy the WHERE clause and (ii) their value for  $R_{vt}$  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 (denoted as *period*, hereinafter) are fetched into main memory. Depending on the values of  $R_{vt}$  and *period*, one of the actions (1)-(3) described in Subsection 4.3 is performed, in order to delete from  $R$  the portion of the tuple which is to be updated. After the designated data have been removed, a tuple is inserted through the system session into a temporary table *update\_temp*. (This table is created through the system session.) The values of the columns of these 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 processed the system session commits, and the rows in *update\_temp* are inserted into  $R$  using the algorithm described in Subsection 4.2.2 for Case (iii). (The step of evaluating the *extended-select* is skipped and *update\_temp* replaces  $T1$ ).

Following the above algorithm, the user session will obtain exclusive locks on the affected rows. If the UPDATE statement contains a WHERE clause, which includes subqueries referencing other tables, the user session will also obtain shared locks on the data used to evaluate the WHERE clause.

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

The algorithm employed for Case (i) above can be used here, modified as follows: (i) A *savepoint* is introduced for the user session at the beginning of the operation. (ii) The resulting tuples are inserted into  $R$  using the algorithm for inserting data into a table for which a key has been defined (Case (iv) of Subsection 4.2.2, with the necessary amendments: The step of evaluating the *extended-select* is skipped. Table *update\_temp* replaces  $T1$ . The savepoint introduced at the start of the operation is used in the case that the database should be rolled back due to the violation of the uniqueness of the primary key). The remarks on lock placement for Case (i) above, also hold for this case, but the remarks made for Case (iv) of Subsection 4.2.2 have to be considered as well.

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

The algorithm described for Case (ii) above can be used for this case, modified as follows: (i) A *savepoint* is introduced for the user session at the beginning of the operation. (ii) The resulting tuples are inserted into  $R$  using the algorithm for inserting data into a table for which a key has been defined (Case (iv) of Subsection 4.2.2, with the necessary amendments). The remarks on lock placement for Case (ii) hold for this case too, but the remarks made for Case (iv) of Subsection 4.2.2 have to be considered as well.

#### 4.4.1. Performance Evaluation

The performance of the update algorithm presented above is evaluated in Figures 7 and 8. Specifically, out of the four distinguished cases for the update of normalised tables, diagrams are presented only for two of them, for brevity reasons. The diagrams for the remaining two cases are similar. Each of the presented diagrams depicts the execution time overhead introduced by the dual session algorithm, as a percentage of the overall execution time. Measurements were obtained for different numbers of affected tuples and various database sizes. The details about the data in the base table, the storage structure, and the implementation platform are those described in Subsection 4.2.3.

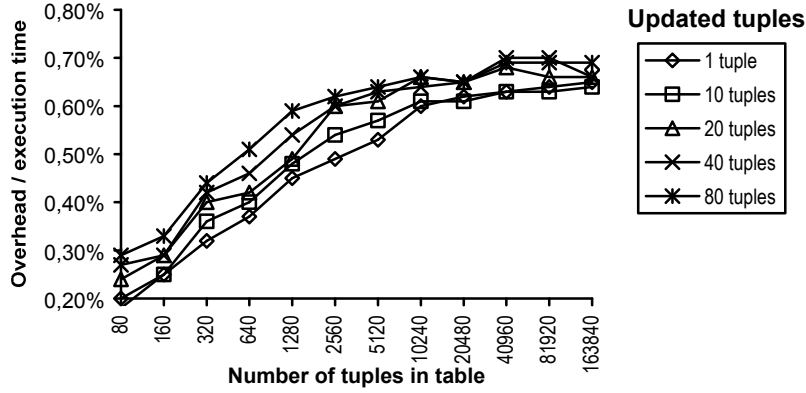


Fig. 7. Updating a table with no key, using the PORTION clause.

The diagrams show that the introduced overhead is extremely low. Our measurements showed that it was less than 0.75% in all cases. This is due to the fact that the only overheads introduced account for the cost of managing the savepoints and the cost of maintaining and switching between two sessions. Contrary to the case of the insertion algorithms (Section 4.2), no extraneous interprocess communication or process switching costs are involved.

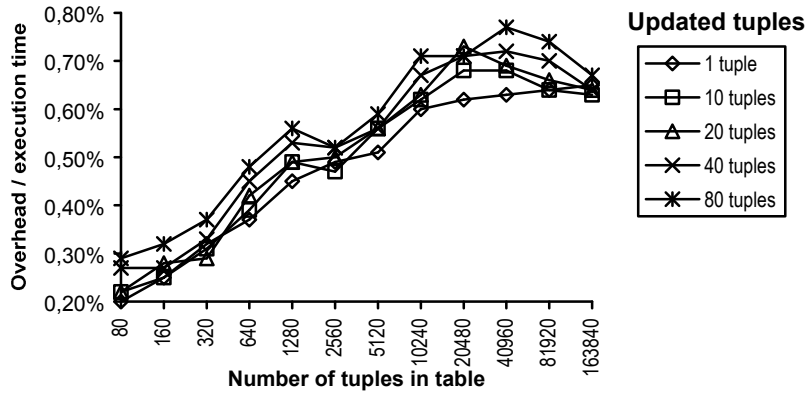


Fig. 8. Updating a table with a key, using the PORTION clause.

## 5. CONCLUSIONS

We presented techniques to support transactions and concurrency control in a layered temporal DBMS. These techniques exploit the transaction support and locking features of the underlying DBMS by the use of a second connection to it, which enables to perform operations on temporary tables. The algorithm design guarantees that no deadlock problems are introduced, due to the fact that locking is done at session level. The overhead introduced by the incorporated techniques is negligible. In particular, our measurements showed that in all cases the increase in execution time was less than 1%. Future work includes support for multiple interval granularities, transaction time and porting of the Temporal Engine on top of object-oriented DBMSs.

*Acknowledgement* – The authors of the paper would like to thank the reviewers for their constructive comments and Ms. Anya Sotiropoulou for her precious assistance.

## REFERENCES

- [1] H. F. Korth and A. Silberschatz. *Database System Concepts*. McGraw-Hill Book Company (1986).
- [2] C. J. Date. *An Introduction to Database Systems, vol. II*. Addison-Wesley Publishing Company (1988).
- [3] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Inc. (1992).
- [4] ORACLE Corporation. *SQL Language Reference Manual (for version 6.0)*. Oracle Corporation (1990).
- [5] Sybase Inc. *Transact SQL User's Guide (for release 4.2)*. Sybase Inc. (1990).
- [6] A. U. Tansel, J. Clifford, S. K. Gadia, S. Jajodia and R.T. Snodgrass (Eds). *Temporal Databases: Theory, Design and Implementation*. Benjamin Cummings (1993).
- [7] G. Ariav. A temporally oriented data model. *ACM Trans. on Database Systems*, **11**(4):499-527 (1986).
- [8] J. Clifford and A. Croker. The historical relational data model (HRDM) revisited. In A. Tansel, J. Clifford, S. K. Gadia, A. Segev, R. Snodgrass, editors, *Temporal Databases: Theory, Design and Implementation*, 6-27, Benjamin Cummings (1993).
- [9] K. Gadia. An homogeneous relational model and query languages for temporal databases. *ACM Transactions on Database Systems*, **13**(4):418-448 (1988).
- [10] McKenzie and R. Snodgrass. Supporting valid time: an historical algebra. *Technical Report TR87-008*. Computer Science Dept., Univ. of North Carolina, Chapel Hill (1987).
- [11] B. Navathe and R. Ahmed. Temporal extensions to the relational model and SQL. In A. Tansel, J. Clifford, S. K. Gadia, A. Segev, R. Snodgrass, editors, *Temporal Databases: Theory, Design and Implementation*, 92-109, Benjamin Cummings (1993).
- [12] R. Snodgrass. The temporal query language TQUEL. *ACM Trans. on Database Systems*, **12**(2):247-298 (1987).
- [13] U. Tansel. An historical query language. *Information Sciences*, **53**:101-133 (1991).
- [14] M. H. Böhlen. Temporal database system implementations. *SIGMOD Record*, **24**(4):53-60 (1995).
- [15] ESPRIT III Project 7224 (ORES). *Deliverable D2: specification of valid time SQL*. April 1993. Available via ftp from the Department of Informatics, University of Athens (<ftp://ftp.di.uoa.gr/pub/ores/reports/d2.ps.gz>).
- [16] N. A. Lorentzos. The interval extended relational model and its application to valid time databases. In A. Tansel, J. Clifford, S. K. Gadia, A. Segev, R. Snodgrass, editors, *Temporal Databases: Theory, Design and Implementation*, 67-91, Benjamin Cummings (1993).
- [17] N. A. Lorentzos and Y. G. Mitsopoulos. SQL extension for interval data. *IEEE Transactions on Knowledge and Data Engineering* **9**(3):480-499 (1997).
- [18] ESPRIT III Project 7224 (ORES). *Deliverable D4.1: implementation of valid time SQL*. April 1993. Available via ftp from the Department of Informatics, University of Athens ([ftp://ftp.di.uoa.gr/pub/ores/reports/d4\\_1.ps.gz](ftp://ftp.di.uoa.gr/pub/ores/reports/d4_1.ps.gz)).
- [19] ESPRIT III Project 7224 (ORES). *Deliverable C3: specification of valid time formalism*. April 1993. Available via ftp from the Department of Informatics, University of Athens (<ftp://ftp.di.uoa.gr/pub/ores/reports/c3.ps.gz>).
- [20] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, **13**(6):377-387, (1970).
- [21] C. Vassilakis, P. Georgiadis and N. Lorentzos. Transaction support in a temporal DBMS. In *Recent Advances in Temporal Databases*, 255-271 Zurich (1995).
- [22] INGRES Corporation. *INGRES SQL and ESQL Reference Manual (for release 6.4)*. INGRES Corporation (1991).