Solving the Phantom Problem by Predicative Optimistic Concurrency Control

Manuel Reimer

ETH Zurich, Institut für Informatik
CH-8092 Zurich, Switzerland
and
Universität Hamburg, Fachbereich Informatik
D-2000 Hamburg 13, Schlüterstr. 70, Fed. Rep. of Germany

Abstract

Database programming languages provide powerful relational structures and operators based on, for example, first-order predicate calculus. Language constructs for database programming, including a transaction concept, require therefore a predicate-oriented approach to concurrency control. A predicative optimistic concurrency control is presented that attacks problems inherent in predicate locking. Only those conflicts that actually occurred between transactions are detected, and well-known query evaluation algorithms are applied instead of algorithms testing the disjointness of certain restricted classes of predicates. For that reason, this approach is an elegant solution to the phantom problem.

1 Introduction: Programmers are Afraid of Phantoms

Data in a database are often subject to integrity constraints that require, one way or another, that statements are executed conditionally. Therefore, a very common situation in *database programming* is the test of preconditions that depend on the actual state of the database prior to actions accessing a database. If preconditions are not fulfilled alternative operations have to be executed indicating some exceptional situation. The following statement sketches such a conditional operation:

```
IF  (• precondition •)
THEN  (• database action •)
ELSE  (• exception •)
```

This work was supported in part by the Deutsche Forschungsgemeinschaft (DFG) under grant no. Schm 450/3 (principal investigator: J. W. Schmidt).

The *semantics* of this conditional statement are precisely defined in sequential programming (cf., e.g., [Grie81]). The database action guarded by the precondition is executed if and only if (iff) the precondition is fulfilled; otherwise, the exception is performed. However, if objects are shared by concurrent programs, additional semantic issues have to be considered. Before or during execution of a guarded command (i.e., database action or exception), the values of shared objects that determine the value of the precondition may be subject to changes by programs running in parallel. Obviously, the value of the precondition may be changed as a consequence, and database actions may have manipulated a database that would not be executed if the precondition would be reevaluated. Therefore, it must be required that the value of the precondition remains constant during the execution of the conditional statement. This requirement must be met if nothing is known in advance about the programs running in parallel [Casa81]. This scenario is assumed throughout the whole paper.

The semantic definition of the conditional statement must capture also the very common precondition testing the existence (respectively the non-existence) of objects in a shared database. Subsequently, the relational approach will be used for data and transaction modelling. With this approach, the precondition may be a membership (non-membership) test for some relation element. The following example will illustrate this kind of precondition. It is based on a library database containing information about lendings of books by persons. Borrowing a book may be modelled through the subsequent conditional statement. (The programming notation adopts concepts of the database programming language *Pascal/R* [Schm77].)

```
IF  (• precondition •)
    NOT SOME le IN lendings ( le.booknr = booktolend )
THEN  (• database action •)
    lendings :+ {[booktolend, borrowingperson]}
ELSE  (• exception •)
    WriteLn ("Book already lent.")
END
```

If two of the above statements referring to the same book are executed concurrently, the scheduler may decide to first test the two preconditions. Both tests are fulfilled if the book is not lent at the beginning. Then, both conditional statements perform their database action. Nevertheless, the final database state will be inconsistent since the same book is lent twice. This

inconsistency resulting from concurrent insertions has been observed as the *phantom problem* (cf. [Eswa76]) and results from instable preconditions.

The notion of a *transaction* refers to a protected simple or composite statement for which the above requirements must be satisfied. A transaction transforms a database from one consistent state to another consistent state ([Eswa76], [Gray81]). In addition, the definition of a transaction specifies the shared database objects that are accessed by the transaction with a certain access mode (read, write, readwrite). The above example can be rewritten by means of the transaction concept introduced in [Mall83].

```
TRANSACTION Borrow
                ( booktolend: . . . ; borrowingperson: . . . );
   IMPORT lendings READWRITE;
BEGIN
   IF  NOT SOME le IN lendings ( le.booknr = booktolend )
   THEN  lendings :+ { [booktolend, borrowingperson] }
   ELSE  WriteLn ("Book already lent.")
   END
END Borrow;
```

Transactions do not operate in general on whole relations (e.g., lendings), but on subrelations fulfilling some selection predicate. The notion of a *selected relation*, introduced in [Mall83] and [Schm83a], defines a *selector* containing a selection predicate and applies selectors to database relations. The example illustrates these concepts.

```
SELECTOR LentBook ( booktolend: . . . ) FOR lrel: . . . ;
BEGIN
   EACH le IN lrel : le.booknr = booktolend
END LentBook;

TRANSACTION Borrow
                ( booktolend: . . . ; borrowingperson: . . . );
   IMPORT sublendings = lendings [ LentBook(booktolend) ]
                        READWRITE;
BEGIN
   IF  sublendings = {}
   THEN  sublendings :+ { [booktolend, borrowingperson] }
   ELSE  WriteLn ("Book already lent.")
   END
END Borrow;
```

With the transaction concept, the requirement for a stable precondition of a conditional statement can now be reformulated. For a transaction to be scheduled consistently, their imported database objects (i.e., selected relations) must remain constant during execution of the transaction.

To solve conflicts raised by the concurrent execution of transactions, accesses to shared objects must be synchronized. *Execution models* have been proposed that guarantee consistent scheduling of concurrent transactions based on various concurrency control methods including predicate locking [Eswa76]. Predicate locking solves the above consistency problems including that of preventing phantoms.

This paper proposes a different concurrency control method, called *predicative optimistic concurrency control.* It is based on the optimistic approach of [Kung81]. Section 2 introduces and defines the idea of the predicative optimistic concurrency control. Its advantages compared to other concurrency control methods are discussed in Section 3. Algorithms for a possible implementation of the predicative optimistic concurrency control are given in Section 4, including optimizations of the proposed method.

2 Predicative Optimistic Concurrency Control

2.1 The Method

The predicative optimistic method to concurrency control is based on the *optimistic assumption* that conflicts between concurrent transactions will occur rather seldom. The method is founded on concepts of the original optimistic approach of [Kung81]. The idea of the *predicative optimistic method* is summarized in the following.

Each transaction is divided into three phases. During the *read phase*, the operations of a transaction are executed. Database objects can be read unrestrictedly, however, write operations are performed on local copies. In the *validation phase*, it is tested if the requirements introduced above are fulfilled. This means that the objects imported by a transaction must not be changed by other transactions. If validation succeeds, the local copies are made global during the *write phase*. Otherwise, the transaction is aborted and restarted.

For each transaction, two sets are maintained: a read set and a write set. The *read set* consists of all selected relations accessed by a transaction. The name of the database relation of which the selected relation is part of and the selector defining the selection predicate are kept in the read set. The *write set* determines the objects written by a transaction. Since selected relations are defined by means of predicates, the objects manipulated through a certain selected relation can be described by the values of the selected relation before and after access. When accessed first, a copy of the selected relation local to the transaction is made, and all further read and write operations are directed to this copy. This means that writes to the global database do not occur during the read phase.

To verify the correct execution of concurrent transactions the criterion of serializability is generally accepted [Eswa76]. Serializability is achieved by assigning unique numbers to transactions and by guaranteeing that whenever $i < j$, then transaction $T(i)$ comes before transaction $T(j)$ in the equivalent serial schedule. The transaction numbers are assigned at the end of the read phase.

Transactions $T(i)$ can be divided with respect to transaction $T(j)$ with $i < j$ into three *classes* ($T(i)$ before $T(j)$ in the equivalent serial schedule):

(1) each $T(i)$ that finishes before $T(i)$ starts;
(2) each $T(i)$ that finishes when $T(j)$ is in its read phase;
(3) each $T(i)$ that finishes after $T(j)$ has begun its validation phase.

Transactions $T(i)$ with $i > j$ don't have to be considered, only the role of $T(i)$ and $T(j)$ has to be changed.

Transactions of class (1) cannot conflict with $T(j)$, so validation against transactions of class (1) is not necessary. For class (2), it must be confirmed that the objects written by $T(i)$ have not been read by $T(j)$. This is checked by testing each write set of $T(i)$ against the read set of $T(j)$ on disjointness. For class (3), the additional requirement has to be met that the objects written by $T(i)$ will not be written by $T(j)$ in parallel. Therefore, each write set of $T(i)$ must be disjoint from the read set *and* from the write set of $T(j)$. If the validation for transaction $T(j)$ fails, $T(j)$ is aborted and restarted from its beginning. If validation succeeds all local copies are transfered to the database in the write phase.

2.2 Formal Definition

Before the predicative optimistic method will be defined precisely, a formal model of the database concepts used in this paper is introduced.

A relational database consists of a set of relation variables. A *relation variable* R has some type $\langle D1, ..., Dd \rangle$. Di is called a *domain*, and d specifies the *degree* of R. A *relation element* of R is denoted by $re(R)$, and it has a value, $\langle v1, ..., vd \rangle$, with vi in Di.

A subset of the universe of R may be assigned to a relation variable R. The *universe* $U(R)$ is defined by $D1 \times D2 \times ... \times Dd$. The *actual value* of relation variable R at some point in *time* is determined by a finite subset $V(R,time) \subseteq U(R)$.

Transactions operate on subrelations fulfilling some selection predicate [Mall83]. *Selected relations* are defined as $SR = \langle R, pred \rangle$; *pred* specifies an arbitrary first-order *predicate* applicable to relation variable R. The universe of SR is defined by

$$U(SR) = \{ re \mid re \in U(R) \text{ and } pred(re) \}$$

and the value of SR at a certain *time* by

$$V(SR,time) = \{ re \mid re \in V(R,time) \text{ and } pred(re) \}.$$

The set $N(SR,time)$ denotes relation elements that are not materialized. It is the set difference between the universe and the actual value :

$$N(SR,time) = U(SR) - V(SR,time).$$

Let T be the set of all active and of all finished transactions t ($t \in T$). A *transaction* consists of a sequence of database operations transforming a database from one consistent state to another consistent state [Gray81]. A transaction operates on selected relations with some access mode specifying the transaction's access right (readonly, writeonly, readwrite) on selected parts of relations. An *access* $a = \langle SR, m \rangle$ defines a selected relation SR that may be accessed through access mode $m \in \{ read, write, readwrite \}$. The set of all accesses of transaction t is given by $A(t)$. The first time at which a transaction performs a certain access a is denoted by $b(a)$ (i.e., beginning of access). Respectively, the last time at which a transaction performs a certain access a is denoted by $e(a)$ (i.e., end of access).

Granting an access to a selected relation SR implies that not only $V(SR,b(a))$, but also $N(SR,b(a))$ may be changed by t. To describe the manipulations on $N(SR,b(a))$ performed by transaction t, a set called *materializations* is introduced :

$$M(SR,t) = N(SR,b(a)) - N(SR,e(a)) = V(SR,e(a)) - V(SR,b(a)).$$

The *phantom problem* arises for transaction t with respect to access a with $a \in A(t)$, iff

$$\exists\, t' \in T-\{t\} \quad (\{ re \mid re \in M(SR,t') \text{ and } re \text{ inserted into } M(SR,t') \text{ between } b(a) \text{ and } e(a) \} \neq \{\}).$$

The *read set* of a transaction t consists of all database objects accessed by t. The objects accessed by t are determined through $A(t)$. For each access $a \in A(t)$, the read set contains $\langle SR, m \rangle$. These read set entries determine $U(SR)$.

The *write set* contains the selected relations written by a transaction t. Written selected relations can be determined by their values at the beginning of access and their values at the end of access. Therefore, the write set contains the elements of $V(SR,b(a))$ and $V(SR,e(a))$ for each access $a \in A(t)$ with access mode different from *read*. Notice, that each access that is element of the write set is also member of the read set.

Validation of transaction t must consider all transactions t' that are members of the classes (2) and (3) introduced in the previous subsection. These transactions are determined by the set

$$T'(t) = \{ t' \mid t' \in T \text{ and } t' \text{ finished between beginning and end of } t \}.$$

Validation of a transaction t is performed only against finished transactions t'. For that reason, the materializations $M(SR',t')$ are known explicitly for each access $a' \in A(t')$:

$$M(SR',t') = V(SR',e(a')) - V(SR',b(a')).$$

With consideration of $U(SR)$ and with the knowledge of the materializations, the following definition of validation for the predicative optimistic approach guarantees serializability. This definition resolves any consistency problems due to concurrent execution of transactions including the phantom problem.

Predicative validation of transaction t succeeds, iff

$$\forall\, a \in A(t) \quad \forall\, t' \in T'(t) \quad \forall\, a' \in A(t') \quad (U(SR) \cap V(SR',b(a')) = \{\} \text{ and } U(SR) \cap V(SR',e(a')) = \{\} \text{ or } m' = read).$$

The objects written by t cannot be in conflict with the objects read by t' since the objects are written by t only on local copies and t' is already finished.

3 Comparison with Other Predicative Concurrency Control Methods

3.1 Preventing Phantoms: Predicate Locking

The concurrency control method most commonly used in database scheduling is locking. To solve the phantom problem *predicate locking* must be used ([Bern81], [Eswa76], [Jord81], [Klug83], [Schl78]). Subsequently, predicate locking is defined in terms of the formal model introduced in Section 2.2.

An access a grants a transaction t to operate on selected relation SR with access mode m. The concurrency control component will lock, accordingly, each element accessible through the selected relation (i.e., each relation element fulfilling the selection predicate). A lock may be granted if and only if there is no other

active transaction holding a conflicting lock. The decision is made at the beginning of the access $b(a)$ performed by t. At this time, the set of active transactions with respect to t and a is determined by:

$$T''(t,a) = \{ t' \mid t' \in T \text{ and } t' \neq t \text{ and } t' \text{ active at } b(a) \}.$$

Since locking must be done before an access to a selected relation is performed, $M(SR,t')$ cannot be determined in advance. $M(SR,t')$ is a subset of $N(SR,b(a))$. Therefore, the whole set $N(SR,b(a))$ must be locked in addition to $V(SR,b(a))$. The concurrency control component may grant a predicate lock on access a to transaction t, iff

$$\forall\, t' \in T''(t,a) \quad \forall\, a' \in A(t')$$
$$((V(SR,b(a)) \cup N(SR,b(a))) \cap (V(SR',b(a)) \cup N(SR',b(a))) = \{\}$$
$$\text{or } (m = read \text{ and } m' = read)).$$

This is equivalent to

$$\forall\, t' \in T''(t,a) \quad \forall\, a' \in A(t')$$
$$(U(SR) \cap U(SR') = \{\} \text{ or } (m = read \text{ and } m' = read)).$$

$U(SR)$ must be constant between $b(a)$ and $e(a)$. This means that the predicate defining SR must be constant between $b(a)$ and $e(a)$. To guarantee this property, additional read locks must be set on all quantified relations occurring in the predicate [Klug83].

The test on disjointness of $U(SR)$ and $U(SR')$ is the basis for the detection of a conflict between two accesses of transactions. Therefore, the predicates defining the selected relations must be tested on disjointness. Unfortunately, this is a hard algorithmic problem, decidable only for certain classes of first-order predicate calculus (cf. e.g., [Eswa76], [Hunt79], [Klug83], [Munz79], [Rose80]). In general, arbitrary selection predicates must be generalized to simpler predicates fitting into some common class of decidable predicates. This may increase the locking granularity and, consequently, may reduce the degree of concurrency.

A serious problem arises through the fact that disjointness tests are based on the *intension* of database operation. Therefore, all possible conflicts are detected independently of their actual occurrence. For that reason, the reachable degree of concurrency may be restricted severely.

Predicate locking is a concurrency control method that applies *prevention* to the phantom problem. By locking the whole set of relation elements possibly fulfilling the selection predicate, conflicts occurring through the materialization of phantoms are completely prevented. This is achieved on the expense of serializing transactions that are not really in conflict.

Apart from the problem of disjointness testing of predicates, the locking approach to concurrency control has some additional disadvantages (cf. [Bada79], [Kung81]):

(1) Setting and releasing of locks is necessary only in cases, when transactions are really in conflict. Locking assumes that conflicts will be frequent. This is called the *pessimistic assumption* of locking.

(2) Each lock request of a transaction requires compatibility tests relative to the locks already set. Accesses to the global data structures representing the locks must be synchronized between parallel transactions. This may heavily reduce the degree of concurrency, even if conflicts do not occur.

(3) All locks must be held until the end of a transaction to enable an independent backup of transactions.

(4) Due to incrementally requesting locks, deadlocks may occur. If deadlocks are prevented by preclaiming of locks, concurrency is heavily reduced.

3.2 The Basic Optimistic Method

The database model used in the original optimistic method of [Kung81] assigns a unique name to each database object. A read set contains the names of all objects accessed by a transaction. In terms of the above model, only $V(SR,b(a))$ is stored in the read set for each selected variable. Therefore, validation in the original optimistic method can be defined as follows.

Validation of transaction t in the original optimistic method of [Kung81] is successful iff

$$\forall\, a \in A(t) \quad \forall\, t' \in T'(t) \quad \forall\, a' \in A(t')$$
$$(V(SR,b(a)) \cap V(SR',b(a')) = \{\} \text{ and } V(SR,b(a)) \cap V(SR',e(a')) = \{\}$$
$$\text{or } m' = read).$$

$V(SR,b(a))$ captures only the relation elements existing at time $b(a)$. Relation elements created by a transaction t are not considered during validation of t. However, the manipulation of $N(SR,b(a))$ performed by t must also be validated. For that reason, the optimistic approach of [Kung81] has inherited the *phantom problem* out of reasons similar to those applying to the physical locking approach.

3.3 Certifying Phantoms: Predicative Optimistic Concurrency Control

Relative to the disadvantages of locking, the optimistic method gains the following advantages:

(1) The optimistic method affects concurrency only if a conflict has really occurred. With the optimistic assumption of a low possibility for conflicts, restarting a transaction is necessary only in the worst case.

(2) Only write sets must be known to all transactions. They must be contained in a global data structure; read sets, however, which are in the average much larger than write sets can be kept locally. Each write set is inspected only once during validation and not with each lock request. These two improvements are resulting in fewer accesses to global data structures thus increasing the degree of concurrency.

(3) Any object may be accessed at any time; the optimistic method does not have the notion of exclusiveness as enforced by exclusive locks.

(4) No transaction is ever waiting, so deadlocks cannot occur. There is, however, the problem of repeated restart of transactions in heavily loaded systems.

In addition to these advantages, the predicative optimistic approach is superior to predicate locking for the following reasons.

(1) The access intension of a transaction is not tested against other intensions, but against the objects actually manipulated by other transactions. Thereby, only *real conflicts* are detected, and fewer transactions are serialized.

(2) Predicates don't have to be tested on disjointness. The tests, $U(SR) \cap V(SR',b(a')) = \{\}$ and $U(SR) \cap V(SR',e(a')) = \{\}$, are an evaluation of the predicate defining *SR* with respect to the sets of relation elements contained in *V(SR',...)*. In other words, the intension is tested against *extensions*. For this test, well-known algorithms can be used that perform an evaluation of first-order predicates. These query evaluation algorithms are obviously available in every relational database system. Therefore, any predicate may be used for the definition of selected relations specifying the accesses of transactions. Neither restricted classes of predicates, nor generalizations of predicates must be introduced as with predicate locking.

In contrast to predicate locking, phantoms are not prevented, but their absence is certified during validation of transactions. This *certification* is the main characteristic of the predicative optimistic approach to concurrency control.

4 Algorithms for Predicative Optimistic Concurrency Control

Subsequently, the algorithms required for the predicative optimistic concurrency control are presented. The notation used for the algorithms is an extension of the programming language Modula-2 [Wirt82] by constructs of the relational database model. Most of the relational constructs are adopted from the database programming language Pascal/R [Schm77]. They consist of a data type *relation* and operators defined on relations, in particular, relation altering operators (i.e., insert (:+), delete (:-)) and logical and relational expressions based on first-order calculus.

Concurrency control requires certain data that must be accessible (i.e., global) to all running transactions. These data structures constitute the module *GlobalData*. It contains the write sets of transactions that are finished or in their write phase. Transaction sets contain information about active transactions (*actives*), about transactions that are in their validation phase (*validaters*), and about transactions that are in their write phase (*writers*). A clock is maintained in *ctime*, and the currently highest transaction identifier is stored in *ctransid*. The body of the module initializes the global data structures.

An entry of the write set consists of fields specifying the transaction responsible for the entry (*transid*; i.e., *t* with respect to the formal definitions of Section 2), the name of the access through which the writes are performed (*accname*; i.e., $a \in A(t)$), the name of the relation manipulated (*relname*; i.e., *R*), the values of the selected relation at the beginning of access (*oldvalues*; i.e., *V(SR,b(a))*), the values at the end of access (*newvalues*; i.e., *V(SR,e(a))*), and the time at which the new values have been written to the database (*time*).

```
MODULE GlobalData;
TYPE
  TransIdent     = CARDINAL;
  TransSet       = SET OF TransIdent;
  GlobalWSEntry  = RECORD
                     transid   : TransIdent;
                     accname   : AccessName;
                     relname   : RelationName;
                     oldvalues,
                     newvalues : Relation;
                     time      : TimeType
                   END;
  GlobalWSType   = RELATION transid, accname OF
                     GlobalWSEntry;
VAR
  globalWS    : GlobalWSType;
  actives,
  validaters,
  writers     : TransSet;
  ctime       : TimeType;
  ctransid    : TransIdent;
BEGIN
  globalWS := {};
  actives := {};
  validaters := {};
  writers := {};
  ctime := StartTime;
  ctransid := 0
END GlobalData.
```

The module *ConcurrencyControl* contains the data structures and algorithms required to control one transaction *t*. Additionally, it operates on the global data structures of module *GlobalData*. An *AccessEntry* represents an access $a \in A(t)$, especially it contains the predicate and the access mode. The predicate set (*predicates*) contains all accesses on which the transaction has operated on. The write set consists of the accesses to which the transaction has written. Write operations are performed on the field *newvalues* of the write set.

```
MODULE ConcurrencyControl;
TYPE
  AccessEntry    = RECORD
                     accname   : AccessName;
                     relname   : RelationName;
                     pred      : Predicate;
                     mode      : (READ, WRITE,
                                  READWRITE)
                   END;
  PredSetEntry   = RECORD
                     accname   : AccessName;
                     relname   : RelationName;
                     pred      : Predicate;
                     time      : TimeType
                   END;
  PredSetType    = RELATION accname OF
                     PredSetEntry;
```

```
  WriteSetEntry = RECORD
                    accname   : AccessName;
                    relname   : RelationName;
                    oldvalues,
                    newvalues : Relation
                  END;
  WriteSetType  = RELATION accname OF
                    WriteSetEntry;
VAR
  predicates : PredSetType;
  writeset   : WriteSetType;
  owntransid : TransIdent;
  begintrans : TransSet;
```

The procedure *BeginTransaction* is executed at the beginning of a transaction. It assigns a new transaction identifier, defines the transaction to be active, and initializes the local data structures. Operations on global data structures that must not be interrupted by parallel operations are guarded by critical sections (denoted by << ... >>).

```
PROCEDURE BeginTransaction;
BEGIN
  <<  ctransid := ctransid + 1;
      owntransid := ctransid;
      begintrans := actives;
      actives :+ {owntransid};  >>
  predicates := {};
  writeset := {}
END BeginTransaction;
```

If a read access is performed by a transaction, the procedure *RequestRead* is executed. It inserts the access specified as parameter into the predicate set in case it is the first operation on this access. The time of this operation is remarked. The subsequent read operation is performed on the database, except if the access is already member of the write set. In that case, the read access operates on the new values contained in the write set entry.

```
PROCEDURE RequestRead (accentry: AccessEntry);
  VAR predentry : PredSetEntry;
BEGIN
  WITH accentry DO
    IF NOT SOME p IN predicates
                (p.accname = accname) THEN
      predentry.accname := accname;
      predentry.relname := relname;
      predentry.pred := pred;
      predentry.time := ctime;
      predicates :+ {predentry}
    END
  END
END RequestRead;
```

The procedure *RequestWrite* inserts an access into the predicate set and into the write set. Thereby, the old values and the new values are evaluated and stored in the write set. Subsequent write operations are directed to the new values of the corresponding write set entry.

```
PROCEDURE RequestWrite (accentry: AccessEntry);
  VAR writeentry : WriteSetEntry;
BEGIN
  WITH accentry DO
    IF NOT SOME w IN writeset
                (w.accname = accname) THEN
      RequestRead(accentry);
      writeentry.accname := accname;
      writeentry.relname := relname;
      writeentry.oldvalues :=
                {EACH r IN relname : pred(r)};
      writeentry.newvalues :=
                {EACH r IN relname : pred(r)};
      writeset :+ {writeentry}
    END
  END
END RequestWrite;
```

The function *NoConflict* is used during validation. It returns the value TRUE, if there is no conflict between a predicate *p* and the entries of transactions *tset* contained in the global write set. The tests, $U(SR) \cap V(SR',b(a'))$ and $U(SR) \cap V(SR',e(a'))$, are performed. A conflict can only occur, if the time of the read access was before the time at that the new values have been written to the database.

```
PROCEDURE NoConflict (p: PredSetEntry;
                      tset: TransSet): BOOLEAN;
BEGIN
<< RETURN NOT SOME w IN globalWS
       ((w.transid IN tset) AND
        (w.relname = p.relname) AND
        (w.time > p.time) AND
        ( SOME e IN w.oldvalues (p.pred(e)) OR
          SOME e IN w.newvalues (p.pred(e)) )) >>
END NoConflict;
```

The procedure *EndTransaction* performs the validation phase, and also the write phase if validation succeeds. The result of validation is returned through the parameter.

```
PROCEDURE EndTransaction (VAR valid: BOOLEAN);
  VAR finishedtrans,
      validatingtrans,
      waittrans       : TransSet;
      conflict        : BOOLEAN;
BEGIN
```

First, the local write set is transfered to the global write set. Since the write operation to the database is performed some time in the future, a default time is remarked.

```
  FOR EACH w IN writeset : TRUE DO
    WITH w DO
      << globalWS :+
           { [owntransid, accname, relname,
              oldvalues, newvalues, MaxTime] } >>
    END
  END;
```

The concurrently running transactions are evaluated to determine the classes of transactions against that validation must be performed. The set *finishedtrans* collects the finished or writing transactions; *validatingtrans* contains the transactions being in their validation phase.

```
<<  begintrans :+ {owntransid+1 .. ctransid};
    validatingtrans := validaters;
    finishedtrans :=
                begintrans - actives + writers;
    validaters :+ {owntransid};  >>
valid := TRUE;
```

Validation against finished transactions is carried out repeatedly until there are no transactions that have finished their validation phase.

```
WHILE (finishedtrans # {}) AND valid DO
  valid := ALL p IN predicates
             ( NoConflict(p, finishedtrans) );
  <<  finishedtrans :=
             validatingtrans - validaters;  >>
  validatingtrans :- finishedtrans
END;
```

Validation of transaction *t* is performed separately against each validating transaction *t'*. If a conflict is detected, *t* is only invalid if *t'* becomes valid. To come to this decision, *t* must await the end of the validation phase of *t'*. The procedure *WaitTrans* delays a transaction until all transactions of *waittrans* have finished their validation phase, and the parameter *valid* returns TRUE if all transactions of *waittrans* are invalid, otherwise FALSE.

```
waittrans := {};
IF valid THEN
  FOR EACH t IN validatingtrans : TRUE DO
    conflict := NOT ALL p IN predicates
                        ( NoConflict(p, {t}) );
    IF conflict THEN waittrans :+ {t} END
  END
END;
IF waittrans # {} THEN
  WaitTrans(valid, waittrans)
END;
```

If validation succeeds, the write phase is executed, otherwise the transaction will be aborted. During writing the changed database parts, the time information is updated in the global write set. The procedures *SendValid* and *SendNonValid* are delivering the corresponding information to the procedure *WaitTrans*.

```
IF valid THEN
  SendValid(owntransid);
  <<  validaters :- {owntransid};
      writers :+ {owntransid};  >>
  FOR EACH w IN writeset : TRUE DO
    (Write w.newvalues to relation w.relname);
    <<  globalWS[owntransid,w.accname].time :=
                                        ctime  >>
  END;
  <<  writers :- {owntransid};
      actives :- {owntransid}  >>
ELSE
  SendNonValid(owntransid);
  <<  globalWS :- { EACH w IN globalWS :
                    w.transid = owntransid };
      validaters :- {owntransid};
      actives :- {owntransid}  >>
END
END EndTransaction;
END ConcurrencyControl.
```

The complete algorithms proposing several additional optimizations can be found in [Bräg83].

5 Concluding Remarks

The paper presents a predicative method to concurrency control that is based on the optimistic approach. Predicative optimistic concurrency control solves the consistency problems due to concurrent execution of transactions including the phantom problem. It has some advantageous properties compared to predicate locking.

The presented method has been implemented at the University of Hamburg in a multi-user database system supporting the implementation of database programming languages [Schm83b]. The system is written in the programming language Modula-2 [Wirt82] and is running on a VAX-11 computer. A second implementation is in development at the ETH Zurich. It will be part of an extension of the personal database system LIDAS [Rebs83] towards a database system operating on a network of personal computers.

The predicative optimistic concurrency control proposed in this paper relies on the optimistic assumption that the probability for conflicting transactions is rather low. This assumption should hold for large databases and transactions that modify only small parts of a database. In environments of frequently conflicting transactions, combined methods should be used that schedule transactions with just the right amount of locking instead of validation [Laus82]. Applied to the approach presented, those selected relations frequently accessed are locked, while those selected relations for which the probability of conflict is low are validated. The scheduler may decide upon the appropriate policy by collected information about the access frequence to certain selected relations or by measuring the predicted selectivity of the selection predicates. Ideas in this direction are proposed in [Bräg83].

Acknowledgments

The author is grateful to J. W. Schmidt and C. A. Zehnder for their continuous support of this work, and to R. P. Brägger, J. Koch, W. Lamersdorf, M. Mall, and P. Putfarken for the encouraging discussions and their comments on earlier versions of this paper.

References

[Bada79]
Badal,D.Z.: Correctness of Concurrency Control and Implications in Distributed Databases. Proc. IEEE COMPSAC Conf., Chicago, November 1979

[Bern81]
Bernstein,P.A., Goodman,N., Lai, M.-Y.: Laying Phantoms to Rest (By Understanding the Interactions Between Schedulers and Translators in a Database System). Harvard University, Aiken Computation Laboratory, Cambridge, 1981

[Bräg83]
Brägger,R.P., Reimer,M.: Predicative Scheduling: Integration of Locking and Optimistic Methods. ETH Zurich, Institut für Informatik, Report 53, 1983

[Casa81]
Casanova,M.A.: The Concurrency Control Problem for Database Systems. Lecture Notes in Computer Science 116, Springer-Verlag, 1981

[Eswa76]
Eswaran,K.P., Gray,J.N., Lorie,R.A., Traiger,I.L.: The Notions of Consistency and Predicate Locks in a Database System. CACM, Vol.19, No.11, November 1976

[Gray81]
Gray,J.N.: The Transaction Concept: Virtues and Limitations. Proc. 7th Conf. on Very Large Data Bases, Cannes, September 1981

[Grie81]
Gries,D.: The Science of Programming. Texts and Monographs in Computer Science, Springer-Verlag, 1981

[Hunt79]
Hunt,H.B., Rosenkrantz,D.J.: The Complexity of Testing Predicate Locks. Proc. Int. Conf. on Management of Data, Boston, May 1979

[Jord81]
Jordan,J.R., Banerjee,J., Batman,R.B.: Precision Locks. Proc. Int. Conf. on Management of Data, Ann Arbor, May 1981

[Klug83]
Klug,A.: Locking Expressions for Increased Database Concurrency. JACM, Vol.30, No.1, January 1983

[Kung81]
Kung,H.T., Robinson,J.T.: On Optimistic Methods for Concurrency Control. ACM TODS, Vol.6, No.2, June 1981

[Laus82]
Lausen,G.: Concurrency Control in Database Systems: A Step Towards the Integration of Optimistic Methods and Locking. ACM Annual Conf., Dallas, October 1982

[Mall83]
Mall,M., Reimer,M., Schmidt,J.W.: Data Selection, Sharing, and Access Control in a Relational Scenario. *In*: Brodie,M.L., Mylopoulos,J.L., Schmidt,J.W. (Eds.): Perspectives on Conceptual Modelling, Springer-Verlag, 1983

[Munz79]
Munz,R., Schneider,H.-J., Steyer,F.: Application of Sub-Predicate Tests in Database Systems. Proc. 5th Conf. on Very Large Data Bases, Rio de Janeiro, October 1979

[Rebs83]
Rebsamen,J., Reimer,M., Ursprung,P., Zehnder,C.A., Diener,A.: LIDAS - The Database System for the Personal Computer Lilith. Proc. INRIA Workshop on Relational DBMS Design, Implementation, and Use on Micro-Computers, Toulouse, February 1983

[Rose80]
Rosenkrantz,D.J., Hunt,H.B.: Processing Conjunctive Predicates and Queries. Proc. 6th Conf. on Very Large Data Bases, Montreal, October 1980

[Schl78]
Schlageter,G.: Process Synchronization in Database Systems. ACM TODS, Vol.3, No.3, September 1978

[Schm77]
Schmidt,J.W.: Some High Level Language Constructs for Data of Type Relation. ACM TODS, Vol.2, No.3, September 1977

[Schm83a]
Schmidt,J.W., Mall,M.: Abstraction Mechanisms for Database Programming. Proc. ACM SIGPLAN Symp. on Programming Language Issues in Software Systems, ACM SIGPLAN Notices, Vol.18, No.6, June 1983

[Schm83b]
Schmidt,J.W., Reimer,M., Putfarken,P., Mall,M., Koch,J., Jarke,M.: Research in Database Programming: Language Constructs and Execution Models. To be published in IEEE Database Engineering

[Wirt82]
Wirth,N.: Programming in Modula-2. Springer-Verlag, 1982