Predicative scheduling
integration of locking and optimistic methods

Report

Author(s):
Brägger, Richard P.; Reimer, Manuel

Publication date:
1983

Permanent link:
https://doi.org/10.3929/ethz-a-000295134

Rights / license:
In Copyright - Non-Commercial Use Permitted

Originally published in:
ETH, Eidgenössische Technische Hochschule Zürich, Institut für Informatik 53

ETH

Eidgenössische Technische Hochschule
Zürich

Institut für Informatik

Richard P. Brägger, Manuel Reimer

Predicative Scheduling: Integration of Locking and Optimistic Methods

July 1983

53

Contents

Predicative Scheduling : Integration of Locking and Optimistic Methods

Richard P. Brägger (*) and Manuel Reimer (*, #)

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. First of all, 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 environments where the optimistic assumption of a low probability for conflicts between concurrent transactions is not met, an integration of selected locking concepts into the predicative optimistic concurrency control is proposed. Finally, predicative scheduling fully integrates locking and optimistic methods. The predicative scheduler decides upon the appropriate concurrency control policy for each database access of transactions. This decision is based on information collected about the frequence of certain accesses or on measuring the selectivity of access predicates.

* ETH Zürich
Institut für Informatik
ETH-Zentrum
CH - 8092 **Zurich**
Switzerland

Universität Hamburg
Fachbereich Informatik
Schlüterstr. 70
D - 2000 **Hamburg** 13
Fed. Rep. of Germany

1 Introduction

A very common problem in *database programming* is the conditional branching to different program parts. This is due to the fact that preconditions must be checked before actions manipulating a database may be executed. If preconditions are not fulfilled, operations are executed alternatively that indicate some exceptional situation. The following conditional statement sketches this programming situation:

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

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 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 stable 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 (tuple). The following example will illustrate this kind of precondition. The example is based on a small university administration database containing information about the schedule of courses held by lecturers. An additional lecture may be scheduled by the subsequent conditional statement. (The programming notation adopts concepts of the database programming language *Modula/R* [Koch83] as introduced in Chapter 3.)

```
IF  (* precondition *)
  NOT SOME s IN schedule
      ((s.lecturer = lecture.lecturer) AND
       (s.datetime = lecture.datetime))
THEN  (* database action *)
  schedule :+ { lecture }
ELSE  (* exception *)
  WriteLn ("Lecturer busy at the given time")
END
```

If two of the above statements are executed concurrently concerning the same lecturer and the same time, the scheduler may select first the preconditions to be tested. Both tests are fulfilled if the lecturer is not busy at the beginning. Then, both conditional statements execute their database action. But, the final database state is inconsistent. This inconsistency resulting from concurrent insertions has been observed as the *phantom problem* [Eswa76]. Therefore, the requirement for a stable precondition must also capture the case of concurrent insertions.

Database programs are perceived subsequently as *transactions* [Gray81]. To solve conflicts raised by the concurrent execution of transactions, accesses to shared objects must be synchronized. *Execution models* are proposed that guarantee the semantic definition of transactions by means of some concurrency control method. A method that solves the consistency problems including the phantom problem is predicate locking [Eswa76].

This paper proposes a different concurrency control method, called *predicative optimistic concurrency control.* It is based on the optimistic approach of [Kung81]. Chapter 2 reviews the original optimistic method. Chapter 3 presents the idea of the predicative optimistic concurrency control and discusses the database programming language Modula/R and its requirements on a predicative concurrency control. The data structures and algorithms required for the predicative optimistic concurrency control are presented in Chapter 4.

Chapter 5 proposes several ideas about the integration of selected locking concepts into the predicative optimistic concurrency control. Preference to read-only transactions by introducing read locks, and avoidance of data handling overhead by means of write locks are two of these methods.

Finally, Chapter 6 presents *predicative scheduling* that fully integrates locking and optimistic methods. The predicative scheduler decides upon the appropriate concurrency control policy for each database access of transactions. Related data structures and algorithms are discussed.

Preliminary investigations in the direction of this paper have been published in [Bräg82] and [Reim82]. The predicative optimistic concurrency control is presented also in [Reim83].

2 The Optimistic Approach to Concurrency Control

Concurrency control is needed to preserve the integrity of shared data. Most current approaches to solve this problem use some kind of *locking* (e.g. [Gray78]). But, there are some serious disadvantages with the locking method:

(1) Locking is necessary only in the case, when transactions are really in conflict. It is assumed that conflicts will be frequent. This is called the *pessimistic assumption* of locking.

(2) Each lock request of transactions requires a comparison with the already set locks on compatibility. Accesses to the global data structure containing the lock information must be synchronized between transactions running in parallel. This may heavily reduce the degree of concurrency, even if conflicts do not occur.

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

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

In [Kung81], a database model is presented where each database object is a node in a tree. In this environment, many transactions fulfill the *optimistic assumptions*:

the number of nodes in the tree is large compared to the number of nodes accessed by all running transactions, and

an often used node is rarely modified.

Whenever these assumptions are met conflicts between concurrently running transactions will occur rather seldom. The idea of the *optimistic method* to concurrency control introduced by [Kung81] can be summarized as follows.

Each transaction is divided into three phases as illustrated by Figure 2.1. During the *read phase,* the operations of a transaction are executed. Objects of the database can be read unrestricted, but writes are performed on local copies. In the *validation phase,* it is tested if the integrity of the database will not be damaged in writing the local copies. If validation succeeds, these copies are made global during the *write phase*. Otherwise, the transaction is aborted and restarted.

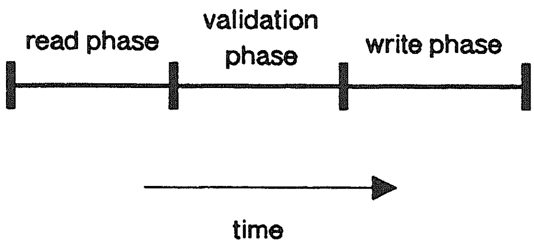

Figure 2.1: Three Phases of a Transaction

Read Phase

The database model used in [Kung81] assigns a unique name to each database object. The concurrency control component manages for each transaction a read set containing the names of all objects read, and a write set containing the names of all objects written by the transaction. At the first write access to a database object, a copy 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.

Validation 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 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 write phase.

Transactions T(i) can be divided with respect to transaction T(j) with i # j into three classes:

(1) each T(i) that finishes before T(j) 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 finished its read phase and T(i) started its validation phase before T(j).

Transactions T(i) that started their validation phase after T(j) must not 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 such transactions 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), it is additionally necessary 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 of transaction T(j) fails, T(j) is aborted and restarted from its beginning.

Two different algorithms for the validation phase are presented in [Kung81]. In the *serial validation algorithm*, the class (3) is empty and must not be considered since at most one transaction is in its validation phase or in its write phase. In the *parallel validation algorithm*, validation phases and write phases are executed in parallel. Therefore, transactions of class (3) must also be considered. The parallel validation algorithm is reviewed in Figure 2.2.

```
valid := TRUE;
FOR EACH t IN class (2) DO
  IF ( write set of transaction t intersects
       read set ) THEN
    valid := FALSE
  END
END;
FOR EACH t IN class (3) DO
  IF ( write set of transaction t intersects
       read set or write set ) THEN
    valid := FALSE
  END
END;
```

Figure 2.2: Parallel Validation Algorithm of [Kung81]

Write Phase

If validation succeeds, all local copies are transfered to the database. These operations are very fast in the special database model of [Kung81] as only pointers must be exchanged.

Some remarks on the parallel validation algorithm:

The two sets representing class (2) and class (3) can be constructed by means of the transaction numbers (details in [Kung81]).

The write set of a transaction must be maintained until all concurrently running transactions have finished their validation phase. But, the concurrency control component may manage only a finite number of write sets. For that reason, a transaction may also be invalidated if it needs an unavailable write set.

The problem of starving transactions, where validation repeatedly fails, can be solved by locking the entire database (a better solution is presented in [Präd82] and in Chapters 5 and 6).

A transaction of class (3) may invalidate the validating transaction even though it becomes invalid itself. A solution to this problem is developed in Chapter 3.

Compared to the disadvantages of locking, the optimistic method gains the following advantages (cf. [Bada79], [Kung81]):

(1) With the optimistic assumption of a low probability for conflicts, aborting a transaction is necessary only in the worst case.

(2) Only the write sets must be known to all transactions. They must be contained in a global data structure, but the read sets, which are much larger than the write sets in the average, can be kept locally. Each write set is observed only once during the 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 a notion of exclusiveness as compared to exclusive locks.

(4) No transaction is ever waiting, so deadlocks cannot occur. There is only the problem of repeatedly aborted transactions in heavily loaded systems.

3 Predicative Optimistic Concurrency Control

Database programming languages provide powerful relational operators (e.g. predicate calculus) for database programming. The semantic definition of language constructs including a transaction concept requires therefore a predicate-oriented approach to concurrency control. This chapter proposes an extension of the optimistic approach towards a predicative optimistic method.

3.1 Database Programming Language Modula/R

The database programming language *Modula/R* [Koch83] integrates the programming language Modula-2 [Wirt82] and concepts of the relational database model [Codd70]. The origin of Modula/R is the database programming language *Pascal/R* which extends Pascal by means of very similar concepts ([Schm77], [Schm80]).

Relations are integrated into Modula/R as a new data type that can be perceived as a set of elements of type *record*. A key of a relation specifies certain fields of the relation element and enforces uniqueness of these fields over all elements of a relation.

The following example illustrates the definition of relation types and the declaration of relation variables in Modula/R. It consists of a small university database.

```
TYPE
  EmployeeRec    = RECORD
                     persNr   : CARDINAL;
                     name     : STRING20;
                     status   : (student, assistant,
                                 professor);
                     salary   : CARDINAL
                   END;

  EmployeeRel    = RELATION persNr OF EmployeeRec;
  CourseRec      = RECORD
                     courseNr : CARDINAL;
                     title    : STRING50
                   END;

  CourseRel      = RELATION courseNr OF CourseRel;

  ScheduleRec    = RECORD
                     courseNr : CARDINAL;
                     persNr   : CARDINAL;
                     time     : [0815..1930];
                     day      : (mon,tue,wed,thu,fri);
                     room     : STRING4
                   END;

  ScheduleRel    = RELATION courseNr, persNr OF
                     ScheduleRec;

VAR
  employees : EmployeeRel;
  courses   : CourseRel;
  schedule  : ScheduleRel;
```

In practice, transactions do not operate on entire database relations. Subsequently, the notion of so-called *selected relation variables* ([Mall83], [Schm83a]) is used to restrict access to selected parts of relations. A selected relation is part of an entire relation variable defined by the restriction that its elements have to fulfill the given selection predicate. To define a specific selection, the notation of a *selector* is used that introduces a selector name (*sp*), binds the selector to a relation type (*RelType*), provides a selection predicate (*p*), and, in addition, may be parameterized:

```
SELECTOR sp (...) FOR frel: RelType;
BEGIN
  EACH r IN frel: p(r,...)
END sp;
```

A selected part of a relation, *rel*, is denoted by *rel[sp(...)]*. Since selected relations are relation variables, each operation defined on relations is applicable to selected relations, as well. Through a selected relation defined by a certain selection predicate, only that part of a database relation can be accessed and modified whose elements fulfill the predicate.

Example: All employees which are professor.

Selector Definition

```
SELECTOR prof FOR erel: EmployeeRel;
BEGIN
  EACH e IN erel: e.status = professor
END prof;
```

Selected Relation

```
employees[prof]
```

Example: All lectures given at the same day at the same time as the lecture, *lect.*

Selector Definition

```
SELECTOR concurrent (se: ScheduleRec)
         FOR srel: ScheduleRel;
BEGIN
  EACH s IN srel: [s.time, s.day] = [se.time, se.day]
END concurrent;
```

Selected Relation

```
schedule[concurrent(lect)]
```

To manipulate database relations, *transactions* must be used. All selected relations that will be manipulated have to be imported in *read*, *readwrite*, or *write* mode. The following example shows the insertion of a new schedule.

```
TRANSACTION InsertSchedule (srec: ScheduleRec);
IMPORT
  lecturer = employees[prof][srec.persNr] READ,
  course   = courses[srec.courseNr] READ,
  parallellectures
           = schedule[concurrent(srec)] READ,
  slecture = schedule[srec.courseNr,
                      srec.persNr] WRITE;
BEGIN
  WITH srec DO
    IF lecturer = VOID THEN
      WriteLn('unknown lecturer')
    ELSIF course = VOID THEN
      WriteLn('unknown course')
    ELSIF SOME l IN parallellectures
                    (l.room = room) THEN
      WriteLn('room occupied')
    ELSIF SOME l IN parallellectures
                    (l.persNr = persNr) THEN
      WriteLn('lecturer busy')
    ELSE slecture := srec
    END
  END
END InsertSchedule;
```

Where are database accesses in this example? In Modula/R, a read access occurs if an imported variable appears in an expression. In the transaction *InsertSchedule,* each if (elsif)-condition contains a read access, for example "*lecturer = VOID*" or "*SOME l IN parallellectures (l.room = room)*". In the first expression, the element of *employees* with the specified *persNr* and *status = professor* is read. The second predicate tests if a lecture in *parallellectures* exists that takes place in the same room at the same time as the lecture *srec*. Each alteration of an imported selected relation must be interpreted as a write access. Such alterations are the operations insert, delete, replace, and assignment with relational operands (e.g., the assignment "*slecture := srec*").

3.2 Read Set and Write Set for Selected Relations

The read sets and write sets have been introduced to detect conflicts in the validation phase. These sets are maintained by the concurrency control component. Before each database access in the read phase, an entry is inserted into the suitable set. The read sets and write sets can be defined as relations. An entry describes an imported selected relation (e.g., *lrel = rel[sp(...)]*) through its local name (*lrel*), its relation name (*rel*), and its selector with the actual parameters (*sp(...)*).

```
TYPE
  SetEntry   = RECORD
                 locName   : RelationName;
                 relName   : RelationName;
                 selector  : SelectorDescriptor
               END;

  SetRelation = RELATION locName OF SetEntry;
```

The Phantom Problem

As observed by [Eswa76], phantoms might occur in a physical locking environment. Physical locking means to lock all elements of a selected relation satisfying the predicate of the selector. But, if existent elements are considered only, another transaction might insert new elements that also fulfill the predicate. These elements are called *phantoms*. To avoid inconsistencies, it is necessary to lock also the non-existent elements. For that reason, predicate locks must be set ([Eswa76], [Bern81], [Klug83]). In the optimistic method, the entries of read sets and write sets correspond to locks (see Chapter 5). Analogously, predicates are used as set entries to avoid phantoms.

Deciding the Conflict between Transactions

The detection of a conflict between two transactions might be based on testing the disjointness of the selected relations imported by the 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], [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 granularity and, consequently, may reduce the degree of concurrency.

A serious problem arises through a certain property of the disjointness test, namely, that it is based on the *intension* of database operation. Conflicts are detected that may occur in principle, but independent from their occurrence in reality. For that reason, the reachable degree of concurrency may be severely restricted.

The following example will illustrate this problem and will suggest a solution. The relation, *employees*, and three different transactions are used.

Relation *employees*

persNr	name	status	salary
10	Miller	assistant	35000
20	Smith	student	5000
30	Brown	assistant	42000
40	Jones	assistant	40000

Transaction T1: All assistants get an additional pay of 10 percent.

Transaction T2: Mr. Smith (persNr = 20) is employed as assistant.

Transaction T3: Mr. Jones (persNr = 40) gets professor.

The read sets and write sets corresponding to these transactions consist of the following predicates.

Read Sets and Write Sets

Transaction T1:	Transaction T2:	Transaction T3:
"status = assistant"	"persNr = 20"	"persNr = 40"

Consider a concurrent schedule with transaction T1 followed by transaction T2 in the equivalent serial schedule. In the validation phase of T2, the read set of T2 and the write set of T1 must be compared. But, it is not decidable by inspecting the predicates if they are disjoint. For that reason, transaction T2 must be aborted, although this situation is serializable. Transaction T2 only read the tuple with *persNr = 20* and with *status = student*. Since the status will be changed not before termination of T1, T2 did not affect the values read by T1. So the two transactions ran without conflict and backup is not necessary.

To detect such situations, the form of the set entries should be changed. After the read phase, each transaction holds not only the predicates, but also the local copies of the written selected relations. Because these copies will not be changed afterwards, they can be used for tests by other transactions. In the validation phase, it is necessary to test whether the reads were affected by writes of earlier transactions. The comparison of predicates is replaced by the test, whether there are values in the local copies of earlier transactions that fulfill the read predicates. To demonstrate this test, the same sample schedule as above is used. Transaction T1 has terminated. Its copy of the selected relation is:

10	Miller	assistant	38500
30	Brown	assistant	46200
40	Jones	assistant	44000

Transaction T2 has started during the write phase of T1. In its validation phase, it is tested whether there exists an element with *persNr = 20* (read predicate) in the local copies of T1. This query returns a negative result, and so T2 can start its write phase.

Another sample schedule will show that not all conflicts can be detected by means of this test. Transactions T3 and T1 are executed in parallel whereby T3 finishes first (T3 < T1). The local copy of this transaction consists of only one element (changed version):

40	Jones	professor	40000

The test in the validation phase of T1 (i.e., "are there tuples with *status = assistant* in a copy?") produces a negative result, meaning that the concurrent execution is correct. But this is wrong. Transaction T1 made its copy before T3 changed the status of Mr. Jones, and therefore gets the status value assistant. Then also the element with *persNr = 40* is included in the local copy of T1. To detect such situations, the old values of the local copies must also be examined (cf. [Eswa76]: "Locks must cover both the old and new values.").

Consequently, the write set entry can be redefined as follows:

```
TYPE
  WriteSetEntry  =  RECORD
                      locName   : RelationName;
                      relName   : RelationName;
                      oldValues : Relation;
                      newValues : Relation
                    END;
```

Selected relations can be imported in mode *read*, *readwrite*, or *write*. If an import has mode *write*, a local copy of the selected relation must be made as with mode *readwrite*. For that reason, a read access occurs with the write predicate which becomes therefore automatically a readwrite predicate. Thus, all predicates can be collected in one set named predicate set. In an additional attribute it is specified whether it is a read or a readwrite predicate (see Section 4.2).

The proposed predicative optimistic method has the following advantages.

(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 must not be tested on disjointness. The test, predicate versus local copy of a selected relation, is performed. 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.

3.3 Validation Phase

Two different algorithms for validation are discussed in [Kung81], a serial and a parallel one. With the relational model, the write phase may last longer than with the model of [Kung81]. For that reason, it is not advisable to perform it serially. Subsequently, the parallel algorithm is examined only.

In this section, the algorithm is improved furtherly. Firstly, concurrent transactions are classified. Then, the necessary tests are evaluated, and finally, timestamps are introduced for validation.

Classification of Concurrent Transactions

In Chapter 2, three classes of concurrent transactions are introduced. Subsequently, this classification is discussed in more detail. For the following, take a transaction T. The beginning of the validation phase of T is chosen to divide the others transactions T' (with T # T') into three groups: transactions that are finished already, transactions that finished their read phase, and all others. These three groups are further divided into six classes as follows and illustrated by Figure 3.1.

T' finished	read phase of T' finished	read phase of T' not finished
(1) before start of T	(3) T' in its write phase	(5) T' in its read phase
(2) after start of T	(4) T' in its validation phase	(6) T' not yet started

Class (1): These transactions did not affect T because they ran before T. They must not be considered during validation of T.
Class (2): These transactions changed selected relations while T was reading, therefore conflicts are possible. This class corresponds with

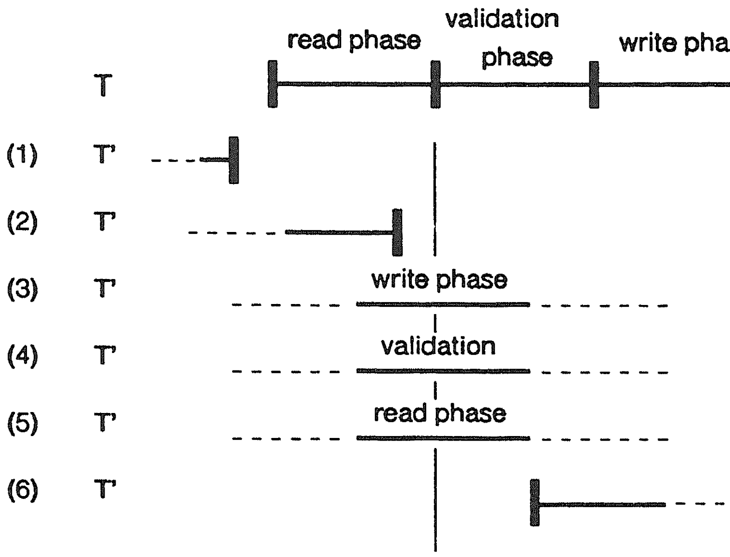

Figure 3.1: Classes of Concurrent Transactions at Beginning of the Validation Phase

class (2) of Chapter 2.

Class (3): These transactions are valid, but are still writing. This class is a part of class (3) of Chapter 2.

Class (4): These transactions started their validation phase before T, but it is possible, that they get invalid later and are aborted. This class is also a part of class(3) of Chapter 2.

Classes (5), (6): These transactions cannot affect T, because in their validation phase conflicts with T are detected by them.

Each transaction has an identification number. These numbers are used to build the classes. Three sets are managed by the concurrency control component: *readers*, *validaters*, and *writers*. These sets collect the numbers of transactions that are in the corresponding phase.

Timestamps

The parallel validation algorithm causes abortions which are not necessary. Consider the example illustrated by Figure 3.2.

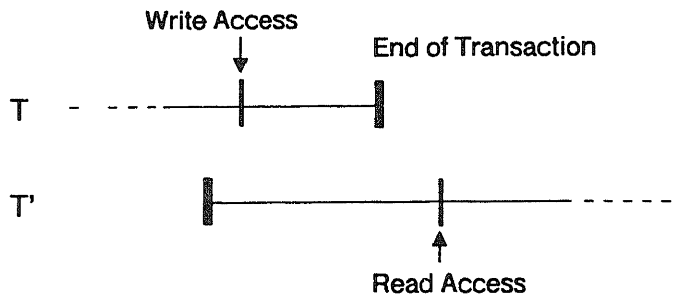

Figure 3.2: Example of Conflicting Transactions

In the validation phase of T, a conflict between T and T' is signaled, although this case is obviously serializable (T' < T). In [Laus82], a method is presented to avoid such abortions. Whenever an object is read, the highest transaction number currently assigned is stored in the read set. In the validation phase of T, it is possible to decide if this object was read after the termination of the conflicting transaction T', because in this case the transaction number of T' is smaller than the number stored in the read set. In this case, a conflict is not signaled. Otherwise, T must be aborted.

Using a clock, this method can be improved furtherly as presented in Figure 3.3.

The current time of the first read access or of the last write access for each selected relation is saved. Thereby, it is decidable if the read access took place really before the write access. This improvement will

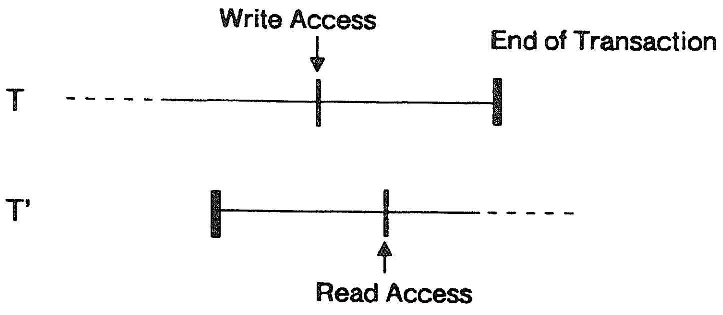

Figure 3.3: Conflict Detection Using a Clock

be described in detail in Section 4.2.

Conditions for Finished Transactions (class (2))

All transactions T' of class (2) are collected in a local set. When transaction T starts its validation phase, the transactions T' are yet terminated correctly and cannot be aborted anymore. Therefore, conflicts between T' and T can only be solved by setting back transaction T. Conflicts can be detected by intersecting the read set of T and the write set of T'.

rw-Test

(1) sets are disjoint
(2) all read times are greater
(3) all read times are smaller
(4) smaller and greater read times

Transaction T' may have read something that transaction T will change (for further discussion see Section 4.2). This requires that T' comes always before T in the equivalent serial schedule. Therefore, cases (3) and (4) cannot be serialized, and T must be aborted. Cases (1) and (2) do not signal a conflict, so T becomes valid.

Conditions for Writing Transactions (class (3))

Transactions of class (3) have finished their validation phase and will not be aborted anymore. It is necessary to test, whether T and T' operated on overlapping selected relations during in the read phase of T (rw-test), and also whether they will conflict during the write phase of T (ww-test). The following cases must be considered.

rw-Test	ww-Test
(1) sets are disjoint	(5) sets are disjoint
(2) all read times are greater	(6) all write times are greater
(3) all read times are smaller	(7) all write times are equal (= default time)
(4) smaller and greater read times	

(*Note*: Not all selected relations are yet written by T'. Therefore, no time is available for such relations. For these relations, a default time which is greater than the time of all other entries is used in the test.)

In case (1), transaction T does not conflict with T'. It directly implies case (5). Also case (2) is consistent, but T' < T is required here. The ww-test would have consequences only in case (7). But this case is impossible here because all write predicates are also read predicates. For that reason, T' must have already written the overlapping selected relation and therefore case (7) is not possible. Case (4) is obviously not serializable, independent of the ww-test; T must be set back.

Case (3) is a bit more complicated. T must come before T' in the equivalent serial schedule. If the write sets are disjoint (5), it could be possible. But it is not known if transaction T' read something which transaction T will update later. This cannot be detected in the validation phase of T'. Since a correct serialization may not be found, T must be restarted.

Conditions for Validating Transactions (class (4))

The difference between class (4) and class (3) is that transactions T' of class (4) can still become invalid. Both the rw-test and the ww-test are necessary. Because none of these transactions has written yet, the timestamps cannot be considered.

rw-Test	ww-Test	Solution
sets disjoint	sets disjoint	any serialization possible
sets disjoint	sets overlap	not possible
sets overlap	sets disjoint	T < T' necessary
sets overlap	sets overlap	backup if T' becomes valid

If the read set of T and the write set of T' are disjoint then any serialization is possible. Each write predicate is also a read predicate and therefore the write set is a subset of the read set. For that reason, the write sets cannot overlap if the read set of T and the write set of T' are disjoint.

In the second case, it is not possible to guarantee that T < T'. Again T' may have read something which T changes afterwards. Thus, T' < T is required. But, T' is still in its validation and can become invalid. In this case, T becomes valid and can starts its write phase. Therefore, T waits for the end of the validation phase of T' and restarts only if T' becomes valid. Cyclic waiting cannot occur because transactions are waiting only for transactions that previously started their validation phase.

Conditions for Classes (5) and (6)

Transactions T' of classes (5) and (6) start their validation phase after T. For that reason, T will be included in classes (1), (2), (3), or (4) of T' and the necessary tests are performed in the validation phase of T'.

Serialization Order

In [Kung81], a transaction number is assigned at the end of the write phase. This number is used to verify the serialization. The end of the write phase determines the serialization order. The various conditions that have to be checked in the validation phase showed that for all transactions T', that started their validation phase before T, $T' < T$ is required (if the extended parallel algorithm is used). Therefore, the point of serialization is the beginning of the validation phase. Consequently, explicit transaction numbers are not necessary any more. Only transaction identifiers are assigned at the start of a transaction.

Integration of Dynamic Aspects

When a transaction T starts its validation phase, it determines the concurrent transactions and classifies them. Since validation phases are executed in parallel, transactions may change their state (i.e., a writing transaction may finish or a validating may begin writing). In the validation phase, these changes are computed after each validation step and a step is repeated until no more changes have occurred.

4 Data Structures and Algorithms for Predicative Optimistic Concurrency Control

This chapter describes all types, variables, and procedures of the concurrency control using the predicative optimistic method. The proposed solution is not directed towards a particular system. Therefore, some types will not be specified in detail. Their meaning can be derived from their name (e.g., TransIdent or RelationName). The notation for the algorithms adopts concepts of the database programming language Modula/R.

4.1 Global Data Structures

Objects that must be accessible to all transactions are collected in a global module. All write sets are saved in the global write set, and a counter for the transaction identifiers and a clock are necessary. The running transactions are maintained in three sets, *actives*, *validaters*, and *writers*.

```
TYPE
  GlobWSEntry = RECORD
                  transId   : TransIdent;
                  locName,
                  relName   : RelationName;
                  oldValues,
                  newValues : Relation;
                  time      : TimeType
                END;

  GlobWSRel   = RELATION transId, locName OF
                  GlobWSEntry;

  TransIdSet  = SET OF TransIdent;
```

```
VAR
  globalWriteSet : GlobWSRel;(* all write sets of
                               transactions that have
                               finished their
                               read phase *)
  cTime        : TimeType;    (* current time *)
  cTransId     : TransIdent;  (* the last assigned
                                 identifier *)

  actives,                    (* all running trans. *)
  validaters,                 (* all Ids of validaters*)
  writers      : TransIdSet;  (* all Ids of writers *)
```

4.2 Local Concurrency Control

Two sets are needed for the validation of a transaction. A predicate set, where the predicates of the imported selected relations are collected, and a write set consisting of all created local copies. An imported selected relation is represented by an import entry.

```
TYPE
  PredSetEntry = RECORD
                   locName,
                   relName : RelationName;
                   pred    : Predicate;
                   time    : TimeType
                 END;

  PredSetRel    = RELATION locName OF PredSetEntry;

  WriteSetEntry  = RECORD
                     locName,
                     relName   : RelationName;
                     oldValues,
                     newValues : Relation
                   END;

  WriteSetRel   = RELATION locName OF WriteSetEntry;
  ImportEntry  = RECORD
                    locName,
                    relName : RelationName;
                    pred    : Predicate;
                    mode    : (READ, WRITE, READWRITE)
                 END;
VAR
  predicates: PredSetRel; (* all used predicates *)
  writeSet  : WriteSetRel;(* local write set with
                             copies *)
  ownTransId: TransIdent; (* transaction identifier *)
  beginSet  : TransIdSet; (* all running transactions
                             at start of transaction*)
```

Procedures

Essentially, the concurrency control component consists of four procedures: *RequestRead*, *RequestWrite*, *BeginTransaction*, and *EndTransaction*. In the following, these procedures are described in detail. The procedure *BeginTransaction* is executed at the beginning of a transaction. If a read access is performed by a transaction, the procedure *RequestRead* is invoked. Respectively, *RequestWrite* is called for write accesses. The procedure *EndTransaction* performs the validation phase, and also the write phase if validation succeeds.

BeginTransaction

The procedure *BeginTransaction* initiates the required sets and local variables. It must not be interrupted by concurrent transactions. Therefore, its operations are guarded by a critical section (denoted by <<.....>>). All running transactions are collected in *beginSet*. A transaction identifier is assigned and the transaction is specified to be active.

```
PROCEDURE BeginTransaction;
BEGIN
  << beginSet := actives;
     cTransId := cTransId + 1;
     ownTransId := cTransId;
     actives :+ {ownTransId}; >>
  predicates := {};
  writeSet := {}
END BeginTransaction;
```

RequestRead

The procedure *RequestRead* controls the read accesses of a transaction. The concerned selected relation is delivered as parameter. The predicate is inserted into the predicate set and the current time is remarked. If the predicate contains a quantifier ranging over some relation then also the range relation must be inserted into the predicate set with a modified predicate (cf. [Klug83] for predicate locking).

No read access occurs in this procedure. The subsequent read operation is performed on the database, with the exception that the selected relation is already member of the write set. In this case, the read operation is executed on the new values contained in the write set entry.

```
PROCEDURE RequestRead (impEntry: ImportEntry);
BEGIN
  IF NOT SOME p in predicates
       (p.locName = impEntry.locName) THEN
     ( insert impEntry into predicates
       and remark cTime )
  END
END RequestRead;
```

RequestWrite

The procedure *RequestWrite* manages the write accesses. The concerned selected relation is inserted into the predicate set and the local copies *oldValues* and *newValues* are created. The question whether the access mode of the selected relation permits a write operation can already be treated during the compilation of the transaction. Therefore no such test is made in *RequestWrite*. Subsequent write operations are directed to the new values of the corresponding write set entry.

```
PROCEDURE RequestWrite (impEntry: ImportEntry);
BEGIN
  IF NOT SOME w in writeSet
      (w.locName = impEntry.locName) THEN
    RequestRead(impEntry);
    ( create local copies );
    ( insert impEntry into writeSet )
  END
END RequestWrite;
```

EndTransaction

The concurrency control of the optimistic method is the validation. The procedure *EndTransaction* manages the necessary tests and also the write phase if the transaction becomes valid. Firstly, the local write set is transfered to the global write set. Because this is not the real write operation to the database, a default time (i.e. the greatest representable time) is remarked. Afterwards, the concurrent transactions are evaluated and two classes are formed. The finished or writing transactions are collected in *finished* and the validating transactions in *validating*.

```
PROCEDURE EndTransaction;
VAR
  finished, validating, waitSet: TransIdSet;
  valid, empty: BOOLEAN;
BEGIN
<< ( insert writeSet into globalWriteSet and
     remark default time );
   validating  := validaters;
   validaters  :+ {ownTransId};
   beginSet    :+ {ownTransId+1,....,cTransId};
   finished    := beginSet - actives + writers;  >>
```

The transactions that started after the validating transaction, but are already finished, are detected by means of *cTransId*. Furthermore, two selectors are introduced.

The selector *PartOf* is used to obtain all entries of the global write set belonging to a set of transactions.

```
SELECTOR PartOf (t: TransIdSet) FOR gWS: GlobWSRel;
BEGIN
  EACH w IN gWS: w.transId IN t
END PartOf;
```

By means of the selector *Intersect*, all entries of the global write set that intersect with a given predicate are selected. The remarked access times are also considered.

```
SELECTOR Intersect (pse: PredSetEntry)
         FOR gWS : GlobWSRel;
BEGIN
  EACH w IN gWS:
    (w.relName = pse.relName) AND
    (w.time > pse.time) AND
    (SOME e IN w.oldValues (pse.pred(e)) OR
     SOME e IN w.newValues (pse.pred(e)))
END Intersect;
```

The validation against finished or writing transactions (i.e. classes (2) and (3) of Section 3.3) can be formulated by means of the selector introduced above. The test is executed in a loop. After each test, a new set of finished transactions is computed. The transactions of set *validating*, which are now finished or started writing, are transfered to set *finished*. Finally, when no more changes happened, one set, *validating*, remains.

```
valid := TRUE;
WHILE valid AND (finished # {}) DO
  valid := ALL p in predicates
               (globalWriteSet[PartOf(finished)]
                              [Intersect(p)] = {});
  << finished := validating - validaters; >>
  validating :- finished
END;
```

Validation against validating transactions must be performed for each transaction separately, because it is possible that the transaction T must await the end of a concurrent transactions T'. If the test detects an intersection then transaction T becomes invalid only if T' becomes valid. Therefore the end of the validation phase of T' must be awaited.

```
waitSet := {};
IF valid THEN
  FOR EACH t IN validating : TRUE DO
    empty := ALL p IN predicates
             (globalWriteSet[PartOf({t})]
                            [Intersect(p)] = {});
    IF NOT empty THEN waitSet :+ {t} END
  END
END;
```

This test is not repeated because transactions starting their validation phase after T are not checked. After this last test, transaction T has to wait eventually for the end of some transactions. If the transaction becomes valid then the write phase follows.

```
IF waitSet # {} THEN WaitValid(valid, waitSet) END;

IF valid THEN
  SendValid(ownTransId);
  << validaters :- {ownTransId};
     writers   :+ {ownTransId};  >>

 (* write phase *)
  FOR EACH w IN writeSet : TRUE DO
   ( write w.newValues to relation w.relName );
    globalWriteSet[ownTransId,
                   w.locName].time := cTime
  END;
  << actives :- {ownTransId};
     writers :- {ownTransId}  >>

ELSE  (* abortion *)
  SendNonValid(ownTransId);
  << globalWriteSet[PartOf({ownTransId})] := {};
     validaters :- {ownTransId};
     actives    :- {ownTransId};  >>

  ( restart transaction )
END

END EndTransaction;
```

Some Remarks

The procedures *WaitValid*, *SendValid*, and *SendNonValid* will not be described in detail. Calling *WaitValid*, the transaction T waits until the transactions in *waitSet* finish their validation phase. If all transactions become invalid, T can start its write phase. But if at least one transaction becomes valid, T must be restarted. The procedures *SendValid* and *SendNonValid* are delivering the corresponding information to the procedure *WaitValid*.

It is not necessary that the predicates in the local predicates set must be deleted if the transaction is restarted. Only the access time must be adjusted.

5 Integration of Selected Locking Concepts

It may seem that the optimistic method, which was transfered to the relational model in Chapter 3, is a rather new approach to concurrency control. Subsequently, the predicative optimistic method is compared to predicative locking. Both approaches use selected relations and the predicates defining the selectors as granules of concurrency control. Therefore, it is reasonable to observe the entries of read sets or write sets as the same concept as locks of predicate locking. This view eases the integration of the optimistic method and locking.

5.1 The Optimistic Method in Locking Terminology

With the locking method, each transaction must set locks on selected relations before accessing them. This can take place freely during execution of the transaction supposed the locks are held until the end of the transaction. The idea of the optimistic method is to collect these locks and to set them altogether. For that reason, three phases are introduced for transactions.

During the read phase, only read accesses to the database are occurring, the writes are performed on local copies. If a read occurs, a read lock is created by inserting the read predicate into the read set. But, all other running transactions must not take notice of the read locks. Therefore, the read set is maintained locally.

The changes must be written to the database after terminating the read phase. Then, all needed write locks are known and can be set altogether. Before that, all read locks must be set with retroactive effect as if they were all set at the start of the transaction. Consequently, it is possible that write locks of other transaction set during the read phase are in conflict with the read locks. This is detected in the validation phase, and perhaps the transaction must be aborted and restarted. If the test succeeds, the transaction tries to set the write locks. If a conflict with an already set write lock appears, the transaction is also aborted. Otherwise, the write phase is executed and all locks are returned afterwards. The optimistic method is therefore only a special view of locking.

Also, the improvements presented in Chapter 3 can be explained by means of this terminology. It is not necessary that the read locks have retroactive effect since beginning of the transaction. A read lock required for the first read access is remark by means of a timestamp in the optimistic method.

The advantages of the optimistic method compared to locking can now be discussed in locking terms.

All necessary comparisons of locks can be performed locally by the transaction. No critical section must be used. This is possible because all needed locks are collected during the read phase. The write locks are inserted into the global write set at start of the validation phase and afterwards, the locks are compared. This is equivalent to preclaiming and consequently deadlocks cannot occur. The tests can be performed in parallel achieving a greater performance if the optimistic assumptions hold.

The read locks can be returned after the test in the validation phase. They must not be kept until the end of the transaction. Using timestamps, the write locks can also be returned just after the end of the write access.

In case of a system error, recovery is very easy. Only transactions interrupted during their write phase have to be considered. All others must only be restarted. Writers have already changed database objects. Recovery must restart their write phase. This imposes that the write set entries must be saved to disk at the beginning of the write phase.

The optimistic method controls concurrency by restarting transactions whereas locking is based on waiting. The following sections are discussing the integration of concepts of predicate locking and of the predicative optimistic method. Basically, the moment of setting locks is investigated. With the basic optimistic method, locks are set altogether at the beginning of the validation phase. The following sections propose some relaxations towards the introduction of read or write locks during the read phase of transactions.

5.2 Preference for Read Transactions

In [Schl81], an extension of the optimistic method is proposed to give preference to read-only transactions in applications where the majority of transactions are queries. The reason for the preference (i.e., query-dominant applications) seems not very sound, since the number of write sets is small in such environments. Therefore, validation of readers can be performed efficiently.

A severe reason to give nevertheless preference to readers is the duration of read-only transactions. They consist in general of complex predicative queries. These queries must be optimized and evaluated by algorithms taking a rather long time. But with increasing duration of a transaction, the probability of a conflict increases. An abortion and restart would make the already performed optimizations and evaluations worthless. For that reason, read-only transactions should be preferred.

The idea is that read-only transactions are setting read locks during their read phase [Reim82]. Consequently, they must not be validated and become always valid; readers are never aborted and restarted.

Conflicts must be detected during validation of writing transactions. It must be tested whether the write locks are in conflict with the read locks of concurrent readers. This is determined by testing the disjointness of the read predicate of the reader and the local copies of the writer (i.e., testing intension vs. extension). In case of conflict, the writer must await the end of the reader for the release of the required lock.

During observation of a reader's predicate set by a concurrent writer, the reader must not insert a new entry into its predicate set (i.e., must not set a lock). This can be guaranteed by delaying the lock request.

5.3 Update in Place

The optimistic method operates with local copies. This may be a disadvantage for large selected relations. A write operation causes always three copy operations. In the read phase, two local copies are prepared, one to save the old values and one for writing. These two copies are needed in any case (i.e., for locking also). If a transaction or system error occurs, they are necessary to recover the database. But, in the optimistic method, a change is performed firstly on the local copy, and in the write phase these values must be copied back to the database. This copy operation may be very costly.

The first idea presented to reduce the expenditure of double writing takes the following transaction model:

transactions operate directly on the database for read and write operations, no local copies are maintained;
read sets and write sets are used;
the system maintains an undo-log to recover the database if a backup occurs;
each transaction is executed in two phases, an update phase and a validation phase.

Update Phase

Reads are executed as with the basic optimistic method. All predicates are collected in a predicate set. Because write operations are also performed directly on the database, no local copies are made. The concurrency control component maintains an undo-log. This undo-log guarantees that all updates can be undone if a transaction is aborted.

Validation Phase

The validation phase is the same as with the basic optimistic method. Only finished and validating transactions must be checked. If a transaction becomes valid, its entries in the undo-log must be deleted. Otherwise all changes of the aborted transaction must be undone. These undo operations must not be performed concurrently to other write or read operations. Transactions having read changed objects of invalid transactions are also invalid and must be restarted.

Discussion

The update in place method avoids double writing. But, the cascading abortion of transactions caused by one invalid transaction excludes this method from application in any realistic environment.

5.4 Write Lock Method

If concurrent transactions may change the same object a non-serializable schedule is resulting in almost every case. Then the effort to restore a consistent database state causes a high expense.

A possible solution is to introduce write locks into the predicative optimistic method. Reads are controlled by means of read sets and a validation phase, writes are controlled applying locks. The same transaction model as in Section 5.3 is used. Instead of write sets, write locks are maintained.

Timestamps

In Section 3.3, timestamps are introduced to distinguish between necessary and unnecessary backups. Writes could occur only once in the write phase. So the end time is required. With the introduction of locks, multiple write operations on the same selected relation are possible. For that reason, the first and the last time when a write access has occurred is remarked. The same access times are remarked for read accesses. These times are used in the validation phase.

Update Phase

Read accesses are treated the same way as with the basic optimistic method. Only write operations are influenced by the concepts of two-phase-locking. Before each write access, a lock must be set. All set write locks are collected in a global write set. It contains predicates, before and after images of the selected relations, and the performed operation. It is necessary to save all changes and not only the old and new values, because each selected relation can be changed more than once during one transaction. To test whether two locks conflict, the before and after images cannot be considered, but the predicates must be tested on disjointness. The locks must be kept until the end of the validation phase. Only if a transaction is already in its validation phase, the before and after images can be considered in the test whether two locks conflict. These tests are performed by the procedure *Disjoint*. If a transaction finishes valid, all its global write set entries are set to be inactive, that means the locks are returned. Possibly, the entries are still needed by concurrently running transactions and therefore cannot be deleted.

```
PROCEDURE RequestWrite (impEntry: ImportEntry):
                                                  BOOLEAN;
BEGIN
  WITH impEntry DO
    IF NOT SOME w IN globalWriteSet
        ((w.transId = ownTransId) AND
         (w.locName = locName)) THEN
       (* insert impEntry as new lock *)
      IF NOT ALL w IN globalWriteSet
          ((w.transId = ownTransId) OR
           w.inactive OR
           Disjoint(w, pred) THEN
        ( Deadlock detection );
        IF deadlock THEN RETURN FALSE END;
        ( wait until lock grantable)
      END;
      ( insert impEntry into globalWriteSet );
      ( remark cTime );
      ( create before image of the old values )
    END
  END;
  RETURN TRUE
END RequestWrite;
```

Validation Phase

Write accesses are serialized, only read operations have to be checked in the validation phase. The point of serialization of transactions is the beginning of the validation phase. Subsequently, two variants of a validation algorithm are discussed for different kinds of transaction environments.

Variant 1

Concurrent transactions are classified as in Chapter 3. One class is formed by all finished transactions, one by all validating, and one by all other transactions. Finished transactions T' require for conflicting database accesses that the first read access of T is greater than the last write access of T'. The test for validating transactions must guarantee that T comes after T' in the equivalent serial schedule. If T has read anything of T', T becomes valid only if T' becomes valid too. Therefore, T has to wait for the end of T'. Running transactions T' must come after T in the equivalent serial schedule. For that reason, the last read times of T must be smaller than the first write times of T'. Changes of the write locks of transaction T' are not dangerous because the above condition is always fulfilled in this case.

To formulate the necessary tests in the validation phase the selector *Intersect* is introduced. It selects all elements of the global write set intersecting a given predicate. The serialization of the compared transactions T and T' can be passed through the parameter *before*.

```
SELECTOR Intersect (pse: PredSetEntry;
                    before: (T'beforeT, TbeforeT'))
         FOR gWS: GlobWSRel;
BEGIN
  EACH w IN gWS:
    (w.relName = pse.relName) AND
    ((before = T'beforeT) OR
     (w.firstTime < pse.lastTime)) AND
    ((before = TbeforeT') OR
     (w.lastTime > pse.firstTime)) AND
    (SOME e IN w.beforeImage (pse.pred(e)) OR
     SOME e IN w.afterImage (pse.pred(e)))
END Intersect;
```

Using this selector, the tests of the validation phase can be expressed as follows.

```
(* all invalid transactions *)
valid := ALL p IN predicates
          (globalWriteSet[PartOf(invalid)]
               [Intersect(p,TbeforeT')] = {});

(* all finished transactions *)
valid := valid AND ALL p IN predicates
           (globalWriteSet[PartOf(finished)]
               [Intersect(p,T'beforeT)] = {});

waitSet := {};
(* all validating transactions *)
FOR EACH t IN validating : TRUE DO
  IF valid THEN
    valid := valid AND ALL p IN predicates
               (globalWriteSet[PartOf({t})]
                   [Intersect(p,T'beforeT)] = {});

    delay := valid AND
             SOME p IN predicates
               SOME w IN globalWriteSet[PartOf({t})]
                 ((w.relName = p.relName) AND
                  SOME e IN w.afterImage (p.pred(e)));

    IF delay THEN waitSet :+ {t} END;
  END
END;

(* all active transactions *)
valid := valid AND ALL p IN predicates
          (globalWriteSet[PartOf(actives)]
              [Intersect(p,TbeforeT')] = {});
```

If validation of transaction T succeeds, T releases all write locks by inactivating the own write set entries. The entries are still needed in the validation phase of concurrent transactions, but the concerned selected relations must not be locked anymore. If T becomes invalid, all transactions that read objects changed by T are caused to become also invalid and must be aborted. In Variant 1, these affected transactions are not aborted immediately. Instead, a test against invalid transactions is executed in the validation phase, and possible conflicts are detected by means of the write set entries of aborted transactions identified by the set *invalid*.

```
IF valid AND (waitSet # {}) THEN
  WaitValid(valid,waitSet)
END;
IF valid THEN
  SendValid(ownTransId);
  ( set globalWriteSet[PartOf({ownTransId})]
    to inactive )
ELSE
  SendNonValid(ownTransId);
  invalid :+ {ownTransId};
  ( set globalWriteSet[PartOf({ownTransId})]
    to inactive );
  ( undo transaction by means of before images
    of globalWriteSet[PartOf({ownTransId})] );
  ( restart transaction )
END;
```

Variant 1 relies on the optimistic assumption of a low probability of conflict. Then the cascading abortion of transactions should occur rarely. If conflicts are frequent, transactions conflicting with invalid ones should be aborted immediately to reduce the cascading dependence of transactions. Therefore, a more pessimistic Variant 2 is introduced as follows.

Variant 2

Variant 2 aborts immediately all transactions T' that conflict with an invalid transaction T. The read predicates are accessible already during the update phase to other transactions. If transaction T becomes invalid, it is set back by undoing its changes in the database. Afterwards, all running transactions are evaluated by means of their read predicates to determine the conflicting transactions T' with respect to the invalid transaction T. These transactions T' must be set invalid and are set back. Since all read predicates of running transactions are globally available, the validation phase must be changed. Only two classes of concurrent transactions, *validating* and *updating* are maintained. Two tests are necessary for both classes, the own read predicates must be compared with the write locks of other transactions and vice versa. To avoid double testing, a set *checked* collecting all executed comparisons is introduced. The write set entries can be deleted after validation.

```
(* all updating transactions *)
FOR EACH t IN updating : TRUE DO
  IF checked[t,ownTransId] = VOID THEN

    v := ALL r IN globalReadSet
                      [PartOfRS({ownTransId})]
                (globalWriteSet[PartOfWS({t})]
                      [Intersect(r,TbeforeT')] = {});
    valid[ownTransId] := valid[ownTransId] AND v;

    v := ALL r IN globalReadSet[PartOfRS({t})]
              (globalWriteSet[PartOfWS({ownTransId})]
                       [Intersect(r,T'beforeT)] = {});
    valid[t] := valid[t] AND v;
    checked :+ {[ownTransId,t]}
  END
END;

(* all validating transactions *)
FOR EACH t IN validating : TRUE DO
  IF checked[t,ownTransId] = VOID THEN

    v := ALL r IN globalReadSet
                      [PartOfRS({ownTransId})]
                (globalWriteSet[PartOfWS({t})]
                      [Intersect(r,T'beforeT)] = {});
    valid[ownTransId] := valid[ownTransID] AND v;

    v := ALL r IN globalReadSet[PartOfRS({t})]
              (globalWriteSet[PartOfWS({ownTransId})]
                       [Intersect(r,TbeforeT')] = {});
    valid[t] := valid[t] AND v;
    checked :+ {[ownTransId,t]}
  END;
```

```
    delay := valid[ownTransId] AND
             SOME r IN globalReadSet
                        [PartOfRS({ownTransId})]
              SOME w IN globalWriteSet
                         [PartOfWS({t})]
                 ((w.relName = r.relName) AND
                  SOME e IN w.afterImage
                             (r.pred(e)));
    IF delay THEN waitSet :+ {t} END
  END;

  IF valid[ownTransId] AND (waitSet # {}) THEN
    WaitValid(valid[ownTransId],waitSet)
  END;
  IF valid[ownTransId] THEN
    SendValid(ownTransId);
    globalWriteSet[PartOfWS({ownTransId})] := {};
    globalReadSet[PartOfRS({ownTransId})] := {}
  ELSE
    SendNonValid(ownTransId);
    ( undo transaction by means of before images
      of globalWriteSet[PartOfWS({ownTransId})] );
    globalWriteSet[PartOfWS({ownTransId})] := {};
    globalReadSet[PartOfRS({ownTransId})] := {};
    ( restart transaction )
  END;
```

Comparing the two variants, it can be observed that the second relies on a more pessimistic assumption. Invalid transactions are aborted immediately, to avoid long chains of cascading abortion of transactions. A disadvantage of Variant 2 is that the read predicates must be kept globally, but changes of the global predicate set by updating transactions after beginning of the validation phase must not be considered. For that reason, updating transactions must not be delayed from inserting new entries into the predicate set.

6 Predicative Scheduling

Predicative scheduling fully integrates locking and optimistic methods. Section 6.1 introduces predicative scheduling, and Section 6.2 describes the data structures and algorithms required for this approach to concurrency control. Section 6.2 is similary structured as Chapter 4 for the predicative optimistic concurrency control.

6.1 Introduction

In [Kung81], the following conclusion is made:

"Consider the case of a database system, where transaction conflict is rare, but not rare enough to justify the use of any of the optimistic approaches. Some type of generalized concurrency control is needed that provides *just the right amount* of locking versus backup."

A first approach in this direction is made in [Laus82]. Transactions using locks and obeying a two-phase-locking protocol are introduced into the optimistic method (so-called *I-type* transactions). To simplify the method, these transactions work also on local copies and require therefore a write phase; but, it is provided that their validation becomes always correct. The serial validation algorithm of [Kung81] is extended. For that reason, the writes to the global database are executed exclusively. Additionally, it must be guaranteed in the validation that the read locks of I-type transactions do not conflict with write set entries of concurrent transactions. If a conflict is detected, the validating transaction is aborted and restarted. In the second approach of [Laus82], transactions that only lock parts of their accessed database objects are introduced towards an attempt of an integration of locking and optimistic methods. This approach is transfered to the predicative concurrency control in the next section. It will be developed furtherly resulting in a deep integration called *predicative scheduling*.

In systems where long lasting transactions are to be executed concurrently with short transactions, predicative scheduling may reduce the number of transaction backups. Long transactions should set locks, and therefore, the probability of a restart is decreased. If a transaction accesses an often changed object, the probability of a conflict is high. Setting a lock can also help in this case. The concurrency control component should keep information of such critical objects in the data dictionary, and should generate locks automatically.

6.2 Data Structures and Algorithms

Transaction Phases

The transaction concept of [Mall83] with import of selected relations is used as for the predicative optimistic concurrency control introduced in Chapter 3. Transactions are divided into three phases: the *operation phase*, the *validation phase*, and the *write phase*. Transactions that only read or lock all imported selected relations do not have a write phase.

Locks for imported selected relations are set implicitly by the concurrency control component depending on information contained in a data dictionary. For this decision, the result procedure *Locking* is introduced returning whether the specified selected relation is to be locked or not.

Non-locked selected relations are treated as in the predicative optimistic method (cf. Chapter 3). Local copies are created for them if they are imported with readwrite or write mode. Modifications of locked selected relations are executed directly (i.e., no local copies are made). The necessary undo and redo information is collected in the *log set*. The different lock modes are treated as introduced in [Reim81] using S-locks for selected relations that are only read, and A-locks and X-locks for the others. If such a selected relation is read, an A-lock is set which is converted to an X-lock at the first write access. To control the write accesses for non-locked selected relations during the write phase, X-locks are set for all write set entries after the test of the validation phase. The compatibility of the different lock modes is explained in Figure 6.1.

lock request	lock already set			
	non	S	A	X
S	+	+	+	–
A	+	+	–	–
X	+	–	–	–
conversion A –> X	+	–	*	*

– : lock request grantable
+ : lock request not grantable
* : this case cannot occur

Figure 6.1: The Compatibility of the Three Lock Modes

To guarantee serializability, two concepts must be integrated. With the two-phase-locking protocol, conflicting transactions are serialized using locks: the first transaction sets a lock and the later must wait. Since transactions can be aborted with predicative scheduling during the validation phase, all write locks must be held until the end of the validation phase. The read locks can be returned at the end of the operation phase because no further changes based on these read objects are executed in the other phases. In the optimistic method of Chapter 3, the beginning of the validation phase is used as point of serialization. Conflicts are solved by restarting an invalid transaction. With predicative scheduling, both concepts must be combined. The beginning of the validation phase must be taken as serialization point since all locks are held at this moment. The end of the validation phase cannot be taken because the read locks are already returned.

Data Structures

The following global data structures must be maintained for predicative scheduling.

```
TYPE

  AccessType     = (S,A,X); (* different lock modes *)

  GlobWSEntry    = RECORD
                     transId   : TransIdent;
                     locName,
                     relName   : RelationName;
                     oldValues,
                     newValues : Relation;
                     lastTime  : TimeType
                   END;

  GlobWSRel     = RELATION transId, locName OF
                    GlobWSEntry;

  LockSetEntry = RECORD
                    transId   : TransIdent;
                    locName,
                    relName   : RelationName;
                    pred      : Predicate;
                    mode      : AccessType;
                    lockTime,
                    unlockTime: TimeType
                  END;

  LockSetRel    = RELATION transId, locName OF
                    LockSetEntry;
  LogSetEntry  = RECORD
                   transId   : TransIdent;
                   locName   : RelationName;
                   opNumber  : CARDINAL;
                   operation : (ins, del, upd);
                   beforeImage,
                   afterImage : RelElementType
                 END;

  LogSetRel    = RELATION transId, locName, opNumber
                   OF LogSetEntry;

  TransIdSet   = SET OF TransIdent;


VAR
  globalWriteSet: GlobWSRel; (* all write sets of
                                transactions that have
                                finished their
                                operation phase *)
  lockSet     : LockSetRel; (* all locked objects *)
  logSet      : LogSetRel;  (* the redo and undo log *)

  cTime       : TimeType;   (* current time *)
  cTransId    : TransIdent; (* the last assigned
                               identifier *)

  actives,                  (* all running
                               transactions *)
  valwriters  : TransIdSet; (* all transIds of writers
                               and validaters *)
```

The following data structures are required locally to each transaction.

```
 TYPE
  PredSetEntry  = RECORD
                    locName,
                    relName   : RelationName;
                    pred      : Predicate;
                    firstTime : TimeType
                  END;

  PredSetRel    = RELATION locName OF PredSetEntry;
  WriteSetEntry = RECORD
                    locName,
                    relName   : RelationName;
                    oldValues,
                    newValues : Relation
                  END;

  WriteSetRel   = RELATION locName OF WriteSetEntry;

  ImportEntry   = RECORD
                    locName,
                    relName   : RelationName;
                    pred      : Predicate;
                    mode      :(READ,WRITE,READWRITE)
                  END;

VAR
  predicates : PredSetRel;  (* all used predicates *)
  writeSet   : WriteSetRel; (* local write set with
                               copies *)
  ownTransId : TransIdent;  (* transaction number *)
  beginSet   : TransIdSet;  (* all running
                               transactions at start
                               of transaction *)
```

Operation Phase

Transactions start with a call of *BeginTransaction*. As in Chapter 4, *BeginTransaction* initialized the required variables. It must not be interrupted by concurrent transactions and is therefore guarded by a critical section. A transaction number is assigned and the transaction is specified as being active.

```
PROCEDURE BeginTransaction;
BEGIN
  << beginSet := actives;
     cTransId := cTransId + 1;
     ownTransId := cTransId;
     actives :+ {ownTransId}; >>
  predicates := {};
  writeSet := {}
END BeginTransaction;
```

Accesses to selected relations imported from the database are controlled using the procedures *RequestRead* and *RequestWrite* (cf. Chapters 4 and 5). To detect conflicting transactions, the following data structures must be maintained: a *predicate set* containing read predicates of non-locked selected relations, a *local write set* collecting the old and new values of non-locked selected relations imported with write or readwrite mode, a *global write set* used in the validation phase, a *global lock set* containing the information about locked selected relations, and a *log set* used as undo-log and redo-log for locked selected relations. Since local copies are created for non-locked selected relations, the operation phase is equivalent to a transaction executed in a two-phase-locking environment. Accesses to non-locked selected relations are treated as local operations and not as database manipulations.

The procedure *RequestRead* controls the read accesses of a transaction. The concerned selected relation is passed as parameter. Read predicates may be inserted already either in the lock set or in the predicate set. If a new entry is necessary then the procedure *Locking* is called that decides whether a lock should be set or not. This procedure will not be described in detail. To detect conflicting locks, the selector *LockConflict* is introduced. The parameter *a* is used to test either still active locks or all defined locks. Furtherly, two procedures are used. The procedure *Overlapping* tests two predicates on disjointness. *Conflicting* examines if two modes are compatible (cf. Figure 6.1).

```
SELECTOR LockConflict (t: TransIdent; r: RelationName;
                       p: Predicate; m: AccessType;
                       a: (allLocks, activeLocks))
         FOR ls: LockSetRel;
BEGIN
  EACH l IN ls : (l.transId # t) AND (l.relName = r)
                 AND Overlapping(l.pred,p)
                 AND Conflicting(l.mode,m)
                 AND ((a = allLocks) OR
                     (cTime < l.unlockTime))
END LockConflict;
```

The procedure *RequestRead* is defined as Boolean result procedure. The value *FALSE* is returned if a deadlock was detected. This can be used during the operation phase to abort a transaction. Read accesses do not occur in this procedure. The subsequent read operations are performed as in Chapter 4.

```
PROCEDURE RequestRead (impEntry: ImportEntry):
                                          BOOLEAN;
VAR
  lockEntry : LockSetEntry;
  predEntry : PredSetEntry;

BEGIN
  WITH impEntry DO
    (* check if already inserted *)
    IF (lockSet[ownTransId,locName] = VOID) AND
       (predicates[locName] = VOID)
    THEN
      IF Locking(ownTransId,impEntry) THEN
        ( assignment of impEntry to lockEntry );
        IF mode = READ THEN lockEntry.mode := S
        ELSE lockEntry.mode := A
        END;
        IF lockSet[LockConflict(ownTransId, relName,
                               pred, lockEntry.mode,
                               activeLocks)] # {}
        THEN
          (* conflicting lock already set *)
          ( Deadlock detection );
          IF deadlock THEN RETURN FALSE END;
          ( wait until lock grantable)
        END;
        lockEntry.lockTime := cTime;
        lockEntry.unlockTime := defaultTime;
        lockSet :+ {lockEntry}
      ELSE
        (* optimistic handling *)
        ( assignment of impEntry to predEntry );
        predEntry.firstTime := cTime;
        predEntry.mode := S;
        predicates :+ {predEntry}
      END
    END
  END;
  RETURN TRUE
END RequestRead;
```

The procedure *RequestWrite* manages the write accesses. Firstly, it checks if the specified selected relation is already locked. In that case, it is possible that the mode must be changed to X. But, such a conversion cannot cause a deadlock. Therefore, deadlock detection is not necessary. The second test (*writeSet[locName] = VOID*) is necessary to decide if an optimistic entry exists already. If an entry does not exist, the procedure *Locking* is called to decide whether locking or non-locking should be applied. In the validation phase, the unlock time is necessary. Consequently, a default time is assigned now (i.e., the greatest representable time). In case of optimistic handling, each write predicate must also be a read predicate. For that reason, an entry of the predicate set is generated additionally.

```
PROCEDURE RequestWrite (impEntry: ImportEntry):
                                          BOOLEAN;
VAR
  lockEntry  : LockSetEntry;
  predEntry  : PredSetEntry;
  writeEntry : WriteSetEntry;
BEGIN
  WITH impEntry DO
    IF lockSet[ownTransId,locName] # VOID THEN
      (* lock already set *)
      lockEntry := lockSet[ownTransId,locName];
      IF lockEntry.mode = A THEN
        lockEntry.mode := X;
        IF lockSet[LockConflict(ownTransId, relName,
                         pred, X, activeLocks)] # {}
        THEN
          ( wait until lock conversion grantable )
        END;
        lockSet :& {lockEntry}
      END
    ELSE
```

```
      IF writeSet[locName] = VOID THEN
        (* impEntry neither set as lock
           nor as optimistic write set entry *)
        IF Locking(ownTransId,impEntry) THEN
          (* new X-lock *)
          ( assignment of impEntry to lockEntry );
          IF lockSet[LockConflict(ownTransId, relName,
                         pred, X, activeLocks)] # {}
          THEN
            ( deadlock detection );
            IF deadlock THEN RETURN FALSE END;
            ( wait until lock grantable )
          END;
          lockEntry.lockTime := cTime;
          lockEntry.unlockTime := defaultTime;
          lockSet :+ {lockEntry}
        ELSE
          (* optimistic handling *)
          IF predicates[locName] = VOID THEN
            (* insert impEntry as read predicate *)
            ( assignment of impEntry to predEntry );
            predEntry.firstTime := cTime;
            predicates :+ {predEntry}
          END;
          predicates[locName].mode := X;
          ( assignment of impEntry to writeEntry );
          ( create local copies );
          writeSet :+ {writeEntry}
        END
      END
    END
  END;
  RETURN TRUE
END RequestWrite;
```

Validation Phase

The procedure *EndTransaction* performs the validation and also the write phase if the transaction becomes valid. Firstly, the local write set is transfered to the global write set. Because this is not the real write operation to the database, a default time (i.e., the greatest representable time) is remarked. The set read locks can be returned immediately. Because the entries are still needed by other transactions, only the unlock time is remarked, but no deletion is made. If a transaction becomes valid, the already set write locks can be returned at the end of the validation phase. But, the proper moment of releasing is the beginning of the validation phase. Therefore, the time of the start of the validation phase is saved and can be used later.

Transactions running in parallel are classified with respect to the moment of serialization (i.e., beginning of the validation phase). The classification of Chapters 3 and 5 are combined. Two different classes can be distinguished: all finished, writing, or checking transactions are collected in *befores*, and all transactions in their read phase in *afters*.

```
PROCEDURE EndTransaction;
VAR
  lockEntry : LockSetEntry;
  befores,                    (* all transactions T'<T *)
  afters    : TransIdSet;     (* all transactions T'>T *)
  begValTime: TimeType;       (* time at start of
                                 validation *)
BEGIN
<< ( insert writeSet into globalWriteSet and
     remark default time );
   begValTime := cTime;
   afters     := actives - valwriters;
   valwriters :+ {ownTransId};
   beginSet   :+ {ownTransId+1,....,cTransId};
   befores    := beginSet - afters;  >>

  (* unlock read locks *)
  FOR EACH l IN lockSet :
      (l.transId = ownTransId) AND (l.mode # X) DO
    l.unlockTime := begValTime;
    lockSet :& {l}
  END;
```

To give an overview of the necessary tests, the two sets *befores* and *afters* of transactions T are divided furtherly. All entries of the predicate set and the write set must be tested on disjointness with the entries of concurrently running transactions T'. Because all set locks are respected by other transactions, locked selected relations must not be considered in the validation phase.

As in Chapter 4, selectors are used to simplify the tests. The selector *PartOfWS* is used to obtain all entries of the global write set belonging to a set of transactions. The selector *PartOfLS* does the same for the lock set.

```
SELECTOR PartOfWS (t: TransIdSet) FOR gWS: GlobWSRel;
BEGIN
  EACH w IN gWS: w.transId IN t
END PartOfWS;

SELECTOR PartOfLS (t: TransIdSet) FOR ls: LockSetRel;
BEGIN
  EACH l IN ls: l.transId IN t
END PartOfLS;
```

By means of the selector *IntersectWS*, all entries of the global write set intersecting a given predicate are selected. The remarked access times are also considered. The selector *IntersectLS* delivers all lock set entries intersecting a given predicate. The log set entries are also considered, because they correspond to the old and new values.

```
SELECTOR IntersectWS (pse: PredSetEntry)
         FOR gWS: GlobWSRel;
BEGIN
  EACH w IN gWS:
    (w.relName = pse.relName) AND
    (w.lastTime > pse.firstTime) AND
    ( SOME e IN w.oldValues (pse.pred(e)) OR
      SOME e IN w.newValues (pse.pred(e)) )
END IntersectWS;
```

```
SELECTOR IntersectLS (pse: PredSetEntry)
         FOR ls: LockSetRel;
BEGIN
  EACH l IN ls:
    (l.relName = pse.relName) AND
    (l.mode = X) AND
    (l.unlockTime > pse.firstTime) AND
    SOME w IN logSet
        ((w.transId = l.transId) AND
         (w.locName = l.locName) AND
         (pse.pred(w.beforeImage) OR
          pse.pred(w.afterImage)))
END IntersectLS;
```

Finished transactions T' of set befores: For these transactions, T after T' in the equivalent serial schedule is required. They are already finished when T starts its validation phase. Therefore, it must be guaranteed that the write operations of T' did not affect the reads of T that were non-locked. Therefore, the write set and the write entries of the lock set of T' must be disjoint from the read set of T. Since all changed values (i.e., old values and new values) of the transactions T' are known, the same test as in Chapter 4 can be applied.

Writing transactions T' of set befores: Again, T after T' is required. But now, T' is still writing when T starts its validation phase. Since the write set is a subset of the read set, the same test as for finished transactions satisfies.

Validating transactions T' of set befores: Since the start of the validation phase is used as serialization point, T after T' is required. The case where T overwrites locked read objects of T' (read locks are returned at the start of the validation phase) is therefore undangerous and does not have to be considered. The contrary case cannot occur, because T' would have to await the end of the read phase of T and therefore would not be in this transaction class. Two different tests are necessary. Firstly, the read set entries of T and the write set entries of T' must not overlap, otherwise T must be restarted. The test, write locks of T' against read set entries of T could be the same as in Chapter 5. Here, the possibility to await the correct end of the validation phase of T' in case of an intersection is not considered furtherly. In such a case, T is also restarted immediately.

Validation against transactions of *befores* can be expressed using the previously defined selectors.

```
valid := ALL p IN predicates
             ((globalWriteSet[PartOfWS(befores)]
                             [IntersectWS(p)] = {})
              AND (lockSet[PartOfLS(befores)]
                          [IntersectLS(p)] = {}));
```

Active transactions in set afters: Now, T before T' in the equivalent serial schedule is required. The test, write locks of T against read set entries of T', is made by T' in its validation phase. Since T' can already have locked something, the read set of T must be compared with these locks. If an intersection results, T must be restarted. Modifications of the database during the write phase and write operations during the read or write phases of concurrent transactions are executed in parallel. To prevent conflicts of such operations, write accesses in the write phase require locks. For that reason, it is checked whether a X-lock can be set for a write predicate. If this lock request cannot be granted, T must be restarted. To prevent that active transactions are setting new conflicting locks, these actions must not be affected and, consequently, must be executed in a critical section.

```
(* validation against 'after' transactions *)
FOR EACH p IN predicates : TRUE DO
  IF valid THEN
   << IF (lockSet[PartOfLS(afters)]
                    [LockConflict(ownTransId, p.relName,
                        p.pred, p.mode, allLocks)] = {})
          AND
          ((p.mode # X)
           OR
           (LockSet[PartOfLS(actives - afters)]
                 [LockConflict(ownTransId, p.relName,
                  p.pred, p.mode, activeLocks)] = {}))
       THEN
         (* valid, set X-lock for write set entry *)
         ( assignment of p to lockEntry );
         lockEntry.mode := X;
         lockEntry.lockTime := cTime;
         lockEntry.unlockTime := defaultTime;
         lockSet :+ {lockEntry}
       ELSE
         valid := FALSE
       END  >>
  END
END;
```

Before T starts its write phase, the write locks set during the read phase can be returned. This is done by assigning *begValTime* to the unlock time of each entry.

```
IF valid THEN
  (* clear all X-locks set during read phase *)
  FOR EACH l IN lockSet[PartOfLS({ownTransId})] :
        (l.mode = X) AND (l.lockTime < begValTime) DO
    l.unlockTime := begValTime;
    lockSet :& {l}
  END
END;
```

Write Phase

The write phase is only executed if T becomes valid. The elements of *newValues* are written to the corresponding database relation. Just after each access, the X-lock can be returned. If the transaction becomes invalid, it is aborted and restarted. All changes made during the read phase are undone using the log set entries. At the end, all sets are updated.

```
IF valid THEN
  (* write phase *)
  (* all write set entries already set as X-locks *)
  FOR EACH w In writeSet : TRUE DO
    ( write w.newValues to relation w.relName );
    globalWriteSet[ownTransId,w.locName].lastTime :=
                                                  cTime;
    lockSet[ownTransId,w.locName].unlockTime := cTime
  END
ELSE
  (* backup *)
  FOR EACH l IN lockSet[PartOfLS({ownTransId})] :
        (l.mode = X) AND (l.lockTime < begValTime) DO
    ( undo transaction by means of beforeImage of
      logSet[ownTransId,l.locName,1] )
  END;
  lockSet[PartOfLS({ownTransId})] := {};
  globalWriteSet[PartOfWS({ownTransId})] := {}
END;
(* update of transaction sets *)
<<  actives :- {ownTransId};
    valwriters :- {ownTransId};  >>
IF NOT valid THEN ( restart transaction ) END

END EndTransaction;
```

7 Conclusions

The methods to concurrency control proposed in this paper are tailored to various application environments. The predicative optimistic concurrency control presented in Chapters 3 and 4 relies on the optimistic assumption of a rather low probability for conflicts between transactions. This assumption should hold for large databases and transactions modifying only small parts of a database. In particular, this method should be well suited for query-dominant applications.

The integration of selected locking concepts discussed in Chapter 5 is directed towards environments with special requirements. If read-only transactions are taking a rather long time due to evaluation of complex queries, they should be preferred by introducing read locks. On the other hand, data handling overhead due to transactions modifying larger database parts can be reduced by means of write locks.

In applications with varying transaction profiles, predicative scheduling should be applied as presented in Chapter 6. Transactions should be scheduled with just the right amount of locking as opposed to validation. Selected relations frequently accessed are locked, but other selected relations for which the probability of conflict is low are validated. The scheduler may decide upon the right strategy by information collected about the frequence of access to specific selected relations or by measuring the predicted selectivity of the selection predicates.

Research must be done for qualitative and quantitative measures to determine the right policy for a specific application and its set of transactions. This will be an object of our further research. Only very few comparisons of different concurrency control strategies are published yet (e.g. [Mena82]).

The predicative optimistic concurrency control and the integration of read locks have been implemented at the University of Hamburg in a multi-user database system supporting the implementation of database programming languages (DBPL Project, [Schm83b], [Reim82]). The system is written in the programming language Modula-2 [Wirt82] and is running on a VAX-11 computer. An implementation of predicative scheduling is under 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.

Acknowledgments

The authors are grateful to J. W. Schmidt and C. A. Zehnder for their continuous support of this work.
The implementation within the DBPL Project is supported partially by the Deutsche Forschungsgemeinschaft DFG, Bonn, Fed. Rep. of Germany (grant Schm 450/3).

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

[Bräg82]

Brägger,R.P.: Prädikative, optimistische Methoden zur Parallelitätskontrolle in Datenbanksystemen. ETH Zurich, Institut für Informatik, Diploma Thesis, September 1982

[Casa81]

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

[Codd70]

Codd,E.F.: A Relational Model of Data of Large Shared Data Banks. CACM, Vol. 13, No. 6, June 1970, pp. 377 - 387

[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, pp. 624 - 633

[Gray78]

Gray,J.N.: Notes on Data Base Operating Systems. Proc. Advanced Course on Operating Systems, Munich, Lecture Notes in Computer Science 60, Springer-Verlag, 1978, pp. 393 - 481

[Gray81]

Gray,J.N.: The Transaction Concept: Virtues and Limitations. Proc. 7th Conf. on Very Large Data Bases, Cannes, September 1981, pp. 144 - 154

[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. ACM SIGMOD Conf. on Management of Data, Boston, May 1979, pp. 127 - 133

[Klug83]

Klug,A.: Locking Expressions for Increased Database Concurrency. JACM, Vol.30, No.1, January 1983, pp. 36 - 54

[Koch83]

Koch,J., Mall,M., Putfarken,P., Reimer,M., Schmidt,J.W., Zehnder,C.A.: Modula/R Report. Lilith Version. ETH Zurich, Institut für Informatik, February 1983

[Kung81]

Kung,H.T., Robinson,J.T.: On Optimistic Methods for Concurrency Control. ACM TODS, Vol.6, No.2, June 1981, pp. 213 - 226

[Laus82]

Lausen,G.: Concurrency Control in Database Systems: A Step Towards the Integration of Optimistic Methods and Locking. Proc. 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

[Mena82]

Menasce,D.A., Nakanishi,T.: Optimistic versus Pessimistic Concurrency Control Mechanisms in Database Management Systems. Information Systems, Vol.7, No.1, 1982, pp. 13 - 27

[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, pp. 426 - 435

[Präd82]

Prädel,U., Schlageter,G., Unland,R.: Einige Verbesserungen optimistischer Synchronisationsverfahren. Proc. GI-Jahrestagung, Kaiserslautern, Springer-Verlag, October 1982, pp. 684 - 698

[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

[Reim81]

Reimer,M., Schmidt,J.W.: Transaction Procedures with Relational Parameters. ETH Zurich, Institut für Informatik, Report 45, October 1981

[Reim82]

Reimer,M.: Grundlagen der Parallelitätskontrolle im DBPL-Projekt. Universität Hamburg, Fachbereich Informatik, DBPL-Memo 111-82, November 1982

[Reim83]

Reimer,M.: Solving the Phantom Problem by Predicative Optimistic Concurrency Control. Proc. 9th Conf. on Very Large Data Bases, Florence, October 1983

[Rose80]

Rosenkrantz,D.J., Hunt,H.B.: Processing Conjunctive Predicates and Queries. Proc. 6th Conf. on Very Large Data Bases, Montreal, October 1980, pp. 64 - 72

[Schl81]

Schlageter,G.: Optimistic Methods for Concurrency Control in Distributed Database Systems. Proc. 7th Conf. on Very Large Data Bases, Cannes, September 1981, pp. 125 - 130

[Schm77]

Schmidt,J.W.: Some High Level Language Constructs for Data of Type Relation. ACM TODS, Vol.2, No.3, September 1977, pp. 247 - 261

[Schm80]

Schmidt,J.W., Mall,M.: Pascal/R Report. Universität Hamburg, Fachbereich Informatik, Bericht Nr.66, January 1980

[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

Berichte des Instituts für Informatik

•Nr.25	U. Ammann:	Error Recovery in Recursive Descent Parsers and Run-time Storage Organization
Nr.26	E. Zachos:	Kombinatorische Logik und S-Terme
•Nr.27	N. Wirth:	MODULA-2
•Nr.28	J. Nievergelt, J. Weydert:	Sites, Modes and Trails: Telling the User of an Interactive System where he is, what he can do, and how to get to places
•Nr.29	A.C. Shaw:	On the Specification of Graphic Command Languages and their Processors
•Nr.30	B. Thurnherr, C.A. Zehnder:	Global Data Base Aspects, Consequences for the Relational Model and a Conceptual Schema Language
•Nr.31	A.C. Shaw:	Software Specification Languages based on regular Expressions
Nr.32	E. Engeler:	Algebras and Combinators
•Nr.33	N. Wirth:	A Collection of PASCAL Programs
•Nr.34	R. Marti, J. Rebsamen, B. Thurnherr:	Meta Data Base Design - Consistent Description of a Data Base Management System
•Nr.35	H.H. Nägeli, R.Schoenberger:	Preventing Storage Overflows in High-level Languages
	J. Hoppe:	A Simple Nucleus written in Modula-2
•Nr.36	N. Wirth:	MODULA-2 (second edition)
Nr.37	Hp. Bürkler, C.A. Zehnder:	EDV-Projektentwicklung - Ein Arbeitsheft für Informatik-Studenten
•Nr.38	H. Burkhart, J. Nievergelt:	Structure-oriented editors
•Nr.39	A. Meier, C.A. Zehnder:	Flächenmodell-Register: Die Strukturen wichtiger geographischer Datensammlungen der Schweiz
•Nr.40	N. Wirth:	The Personal Computer Lilith
Nr.41	T.M. Fehlmann:	Theorie und Anwendung des Graphmodells der Kombinatorischen Logik
Nr.42	E. Graf:	Probabilistische Algorithmen und Computer-unterstützte Untersuchungen von probabilistischen Primalitätstests
•Nr.43	H. Burkhart:	Konzepte zur Systematisierung der Benutzerschnittstelle in interaktiven

		Systemen und ihre Anwendung auf den Entwurf von Editoren
•Nr.44	J. Nievergelt, F.P. Preparata:	Plane-sweep Algorithms for Intersecting Geometric Figures
Nr.45	M. Reimer, J.W. Schmidt:	Transaction Procedures with Relational Parameters
Nr.46	J. Nievergelt, H.Hinterberger, K.C. Sevcik:	The Grid File: An adaptable, symmetric multi-key file structure
Nr.47	J. Nievergelt:	Errors in dialog design and how to avoid them
Nr.48	P. Läuchli:	PG - Ein interaktives System für die Manipulation von Figuren der projektiven Geometrie
Nr.49	A. Meier:	A Graph Grammar Approach to Geographic Data Bases
Nr.50	J. Rebsamen, M. Reimer, P. Ursprung, C.A. Zehnder:	LIDAS A Database System for the Personal Computer Lilith The Database Management
Nr.51	K.J.Lieberherr, S.E. Knudsen:	Zeus: A Hardware Description Language for VLSI
Nr.52	F.L.Ostler:	An SMD Disk Controller for the Lilith Computer
Nr.53	R.P.Brägger, M.Reimer:	Predicative Scheduling: Integration of Locking and Optimistic Methods

• out of stock