Acta Cybernetica 14 (2000) 419-453.

Hybrid Concurrency Control and Recovery for Multi-Level Transactions

Klaus-Dieter Schewe * Torsten Ripke * Sven Drechsler *

Abstract

Multi-level transaction schedulers adapt conflict-serializability on different levels. They exploit the fact that many low-level conflicts (e.g. on the level of pages) become irrelevant, if higher-level application semantics is taken into account. Multi-level transactions may lead to an increase in concurrency.

It is easy to generalize locking protocols to the case of multi-level transactions. In this, however, the possibility of deadlocks may diminish the increase in concurrency. This stimulates the investigation of optimistic or hybrid approaches to concurrency control.

Until now no hybrid concurrency control protocol for multi-level transactions has been published. The new FoPL protocol (**F**orward **o**riented Concurrency Control with **P**reordered **L**ocking) is such a protocol. It employs access lists on the database objects and forward oriented commit validation. The basic test on all levels is based on the reordering of the access lists. When combined with queueing and deadlock detection, the protocol is not only sound, but also complete for multi-level serializable schedules. This is definitely an advantage of FoPL compared with locking protocols. The complexity of deadlock detection is not crucial, since waiting transactions do not hold locks on database objects. Furthermore, the basic FoPL protocol can be optimized in various ways.

Since the concurrency control protocol may force transactions to be aborted, it is necessary to support operation logging. It is shown that as well as multi-level locking protocols can be easily coupled with the ARIES algorithms. This also solves the problem of rollback during normal processing and crash recovery.

Keywords. [H2.4] Transaction Processing, Concurrency [H2.7] Logging and Recovery

General Terms. Algorithms, Reliability

*Computer Science Institute, Clausthal Technical University, Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, FRG [schewe|ripke|sdrechsl]@informatik.tu-clausthal.de

1 Introduction

One of the major intentions underlying the development of database systems was data sharing. As a consequence user programs must be realized as atomic units, which leads to the well-known notion of a *transaction*. Roughly spoken a transaction is the sequence of database operations resulting from program execution. Although these sequences must be interleaved to achieve acceptable performance, the effect must be the same as if transactions were executed sequentially.

Transaction throughput is a crucial issue for all databases. The common approach in practice considers conflict-serializable schedules, where conflicts correspond to read- and write-operations on database objects [8, 20]. No matter which granularity is taken for these objects – pages, records or even relations occur in practice – this approach rules out acceptable, but formally not serializable schedules.

In order to increase the rate of concurrency *multi-level transactions* (as a special form of nested transactions) have been introduced. They already occurred in System/R supporting both short-time locking on pages and locking on records [17]. A general theory of multi-level transactions has been developed in [1] and extended to a discussion of suitable protocols in [23, 24]. The basic idea of multi-level conflict-serializability is that sequences of low-level, e.g. page-level, database operations represent application-dependent operations on higher levels, and there are usually less conflicts on higher levels. Consequently, some of the conflicts on lower levels may be ignored. We shall present the gist of the multi-level transaction model in Section 2. In this context we also extend some notions of basic serializability theory to the case of multi-level transactions. These notions consider recoverability, cascade-freeness and strictness.

In distributed databases multi-level transactions occur naturally [4, 19]. E.g., in distributed object bases we may think of a global level, a local logical object level, a local level of physical objects and a page level. This is the view adopted in the DOMOCC project currently under investigation at Clausthal.

The general approach to concurrency control is the use of locking protocols, especially two-phase locking [20]. It will be shown how to generalize lock protocols to multi-level transactions. This will fill Section 3. The major problems with this approach are transaction throughput and the possibility of deadlocks due to transactions waiting for each other to release locks. There are several algorithms for deadlock detection in distributed databases with non-negligible complexity, e.g. [5, 18]. In addition, in interactive systems or applications with long-term transactions, waiting for the release of any lock may be not acceptable.

Therefore, alternatives to locking protocols dominate the research in concurrency control. The solutions comprise timestamp protocols [13, 14], optimistic protocols [2, 7, 9, 12] and hybrid protocols [3, 10, 11] combining at least two of the other approaches. Unfortunately, none of the existing optimistic or hybrid concurrency control protocols has been generalized to multi-level transactions so far. For example, the *optimistic dummy lock* (ODL) protocol [11] is basically organized as an optimistic scheduler using read/write-labels instead of locking objects.

Then certification tests for the existence of these labels and the final write phase locks objects to be updated. Unfortunately, a direct generalization to multi-level transactions is not possible.

In this paper we present a new hybrid protocol called FoPL (*forward oriented concurrency control protocol with preordered locking*), which is a provably correct protocol for multi-level transactions [21]. FoPL exploits that multi-level schedulers can be composed from schedulers for each of the involved levels [23, 24]. Then the ODL idea is refined such that access lists are defined for all such levels. More precisely, labels are kept in a list according to the time points when they have been set. Commit handling then requires the labels of a validating transaction to be shifted to the head of the list. In contrast to ODL the new FoPL protocol will use forward oriented validation. FoPL will be presented in detail in Section 4.

When combined with queueing and deadlock detection, the protocol is not only sound, but also complete for multi-level serializable schedules. This is definitely an advantage of FoPL compared with locking protocols. The complexity of deadlock detection is not crucial, since waiting transactions do not hold locks on database objects.

Given the basic FoPL protocol we are able to discuss several optimizations. These comprise a more optimistic locking strategy, the processing of earlier or partial rollbacks, and specific capabilities related to absorption. Section 5 is devoted to the discussion of these extensions.

In this context we start with initial considerations concerning the comparison of FoPL with locking protocols. We focus on implementation costs and transaction throughput. This will be done in Section 6.

Since the concurrency control protocol may force transactions to be aborted, it is necessary to support operation logging. For this the sophisticated ARIES algorithms [16, 22] are generally accepted as a good starting point. We show how to extend the algorithms to multi-level transactions, both for locking protocols and FoPL. This also solves the problem of rollback during normal processing and crash recovery. The extension called ARIES/ML [6] also enhances the work by Lomet [15]. The solution to recovery will be presented in Section 7. We conclude with a short summary.

2 The Multi-Level Transaction Model

A multi-level transaction is a special kind of an open nested transaction, where the leaves in the transaction tree have the same depth. Each node in the tree corresponds to some operation implemented by its successors. The root is a transaction. The lowest level L_0 corresponds to operations that access directly the physical database. Therefore, we first define the operations of a multi-level system.

Definition 1 An *n-level-system* $\mathfrak{L}$ consists of n levels $L_i = (\mathfrak{D}_i, \mathfrak{F}_i)$ $(i = 0, \ldots, n-1)$, where $\mathfrak{D}_i$ is a set of *objects* and $\mathfrak{F}_i$ a set of *operators*. An L_i*-operation* is an element of $\mathcal{O}_i = \mathfrak{F}_i \times \mathfrak{D}_i$. □

We write $\mathfrak{L} = (L_{n-1}, \ldots, L_0)$. The levels are numbered in a bottom-up manner.

Example 1 In the DOMOCC project at Clausthal Technical University we investigate distributed object bases. For these it is imaginable to use a 4-level-system. The highest level L_3 should correspond to global logical objects, the next lower level L_2 to local logical objects associated with a unique site, level L_1 to local physical objects, i.e. records, and finally L_0 should correspond to the page level.

Then operations on L_3 as defined before schema fragmentation will be implemented by operations on L_2, these again by operations on the record level L_1, which finally give rise to reading and writing pages of the physical store. □

2.1 Multi-Level Transactions

An n-level transaction is defined next exploiting the notion of an *index tree*, which is a finite set of finite sequences over $\mathbb{N} - \{0\}$. We let $(\mathbb{N} - \{0\})^*$ denote the set of all such sequences. $\mid \alpha \mid$ denotes the length of $\alpha \in \mathbb{N}^*$. Furthermore, we identify numbers with sequences of length 1 and denote the empty sequence by ϵ.

As a syntactic convention we shall use small Greek letters $\alpha, \beta, \mu, \nu, \ldots$ for such number sequences and small Latin letters $i, j, k, \ell, \ldots$ for the numbers in these sequences.

Definition 2 An *index tree* of depth n is a finite subset $I \subseteq (\mathbb{N} - \{0\})^*$ with

- $\epsilon \in I$,
- $\alpha(k+1) \in I \Rightarrow \alpha k \in I$ and
- $\alpha \in I \wedge \mid \alpha \mid < n \Leftrightarrow \alpha 1 \in I$

for all $\alpha \in (\mathbb{N} - \{0\})^*$ and $k \in \mathbb{N}$.

An *n-level-transaction* T_j consists of

- an index tree I of depth n,
- a mapping which assigns to each $\alpha \in I$ an $L_{n-|\alpha|}$-operation, denoted as $o_{j\alpha}$ and
- partial orders $<_i^{(j)}$ on each $\mathfrak{O}_i^{(j)} = \{o_{j\alpha} \mid \mid \alpha \mid + i = n\}$, such that $o_{j\alpha k} <_i^{(j)} o_{j\beta\ell} \Rightarrow k < \ell$ holds.

We call $<_i^{(j)}$ the $L_i^{(j)}$*-precedence relation* of the transaction T_j. □

By abuse of notation we shall talk of the transaction T_j over the index-tree I. Furthermore, we write $op_{j\alpha}(x)$ for the operation $o_{j\alpha} = (op, x)$. In order to have a uniform notation for all levels we also allow to write o_j for T_j.

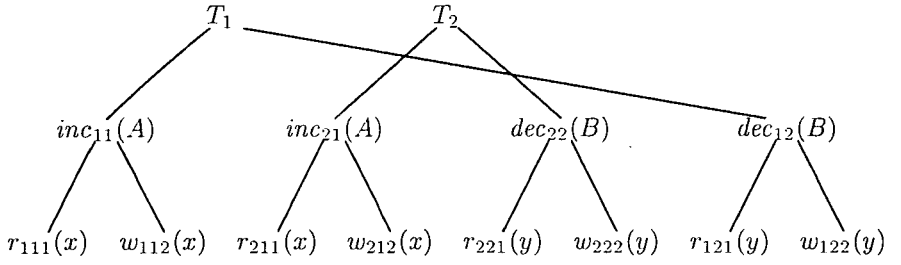

Figure 1: Serializable multi-level schedule

Since precedence relations are meant to express a necessary ordering of implementing operations it is natural to require

$$o_{j\alpha} <_i^{(j)} o_{j\beta} \Leftrightarrow o_{j\alpha k} <_{i-1}^{(j)} o_{j\beta\ell} \quad \text{for all } k \text{ and } \ell, \tag{1}$$

whenever the involved operations exist. In this case, the transaction T_j is *well-defined*. In the sequel we shall tacitly assume that all transactions are well-defined.

Example 2 The trees rooted at T_1 and T_2 in Figure 1 define two 2-level-transactions over the same index tree $I = \{\epsilon, 1, 2, 11, 12, 21, 22\}$. Here w and r correspond to read- and write-operations, *inc* and *dec* to incrementation and decrementation. Thus, we may assume $<_0^{(j)}$ to be defined by

$$r_{111}(x) <_0^{(1)} w_{112}(x) \qquad \text{and} \qquad r_{121}(y) <_0^{(1)} w_{122}(y)$$

and $<_1^{(1)}$ as being empty.

Analogously, define $<_0^{(2)}$ by

$$r_{211}(x) <_0^{(1)} w_{212}(x) \qquad \text{and} \qquad r_{221}(y) <_0^{(1)} w_{222}(y)$$

and let $<_1^{(2)}$ be empty.

However, if we claimed also $w_{112}(x) <_0^{(1)} r_{121}(y)$ – i.e., $<_0^{(1)}$ is total – then the well-definedness condition (1) would imply $inc_{11}(A) <_1^{(1)} dec_{12}(B)$. □

The edges in a transaction tree represent the implementation of a L_i-operation by a set of L_{i-1}-operations. If $o_{j\mu k}$ is an L_i-operation of a transaction T_j, then $trans(o_{j\mu k}) = o_{j\mu}$ $(0 \le i < n)$ is the L_{i+1}-operation that invokes $o_{j\mu k}$. In particular, for $i = n-1$, i.e. μ is empty, we get $trans(o_{jk}) = T_j$. Conversely, $act(o_{j\nu}) = \{o_{j\nu\ell} \mid \nu\ell \in I\}$ defines the set of L_{i-1}-operations implementing the L_i-operation $o_{j\nu}$.

More generally, for $i' > i$ we may define iteratively the $L_{i'}$-operation that indirectly invokes an L_i-operation $o_{j\mu}$ by

$$trans_{i'}(o_{j\mu}) = trans^{i'-i}(o_{j\mu}) \quad . \tag{2}$$

Note that $i' = i+1$ leads to the direct predecessor in the transaction tree as defined by *trans*.

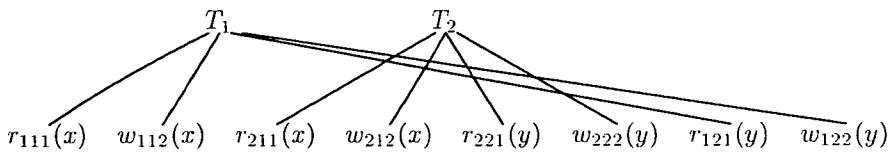

Figure 2: Non-serializable 1-level-schedule

Conversely, for an L_i-operation $o_{j\nu}$ let $act_{i-1}(o_{j\nu}) = act(o_{j\nu})$ and

$$act_{i'}(o_{j\nu}) = \bigcup_{o_{j\nu k} \in act(o_{j\nu})} act_{i'}(o_{j\nu k}) \qquad \text{for } i - i' > 1, \tag{3}$$

i.e. $act_{i'}(o_{j\nu})$ denotes the set of $L_{i'}$-operations implementing $o_{j\nu}$ indirectly through several levels.

Example 3 Consider again transaction T_1 in Figure 1. Here we have

$$\begin{aligned} act(T_1) = act_1(T_1) &= \{inc_{11}(A), dec_{12}(B)\} \quad , \\ act_0(T_1) &= \{r_{111}(x), w_{112}(x), r_{121}(y), w_{122}(y)\} \quad , \\ act(inc_{11}(A)) = act_0(inc_{11}(A)) &= \{r_{111}(x), w_{112}(x)\} \end{aligned}$$

and

$$\begin{aligned} trans(inc_{11}(A)) = trans_2(inc_{11}(A)) &= T_1 \quad , \\ trans(w_{112}(x)) = trans_1(w_{112}(x)) &= inc_{11}(A) \quad , \\ trans_2(w_{112}(x)) &= T_1 \quad . \end{aligned}$$

□

2.2 Multi-Level Schedules

The execution of concurrent transactions is described by an n-level-schedule. These are illustrated by forests in Figures 1 and 2.

Definition 3 For a set $\mathfrak{O}_n = \{T_1, \ldots, T_k\}$ of n-level-transactions let $\mathfrak{O}_i = \bigcup_{j=1}^{k} \mathfrak{O}_i^{(j)}$ be the set of all L_i-operations in these transactions ($0 \leq i < n$). Then an *n-level-schedule* on $\mathfrak{O}_n$ is given by a partial order $<_0$ on $\mathfrak{O}_0$ containing all $L_0^{(j)}$-precedence relations. □

We write $S = (\mathfrak{O}_n, \mathfrak{O}_{n-1}, \ldots, \mathfrak{O}_0, <_0)$ for such a schedule defined on $\mathfrak{O}_n$. Then $<_0$ induces a partial order $<_i$ on each level by

$$o_\mu <_{i+1} o_\nu \Leftrightarrow \forall o_{\mu k} \in act(o_\mu). \forall o_{\nu \ell} \in act(o_\nu).\ o_{\mu k} <_i o_{\nu \ell} \quad . \tag{4}$$

Using this, we may define the *level-by-level schedule* $S_{i,j}$ ($j < i \leq n$) as the one-level-schedule $(\mathfrak{O}_i, \mathfrak{O}_j, <_j)$.

Example 4 The schedule in Figure 2 is the level-by-level schedule $S_{2,0}$ of the one in Figure 1. We dispense with a discussion of how to reorganize the underlying index-trees. □

The well-definedness assumption for transactions implies two simple properties as shown in the next lemma. The first one was originally used in [24] to define the partial order $<_i$ on level L_i. The second property is the plausible *conformity*-condition from [21]. Informally, it states that whenever two operations in some transaction have to occur in a certain order, then they must do so in every schedule.

Lemma 1 *1. For any two L_i-operations o_μ, o_ν in a n-level-schedule S we have*

$$o_\mu <_i o_\nu \Leftrightarrow \forall o_{\mu\varrho} \in act_0(o_\mu).\forall o_{\nu\sigma} \in act_0(o_\nu).\ o_{\mu\varrho} <_0 o_{\nu\sigma} \quad . \tag{5}$$

2. For each n-level-schedule S we have $<_i^{(j)} \subseteq <_i$ for all i and j.

Proof. For the proof of (i) we proceed by induction on i. For $i = 1$ the claimed equivalence in (5) is just the definition (4). For $i > 1$ we have

$$o_\mu <_i o_\nu \Leftrightarrow \forall o_{\mu k} \in act_{i-1}(o_\mu).\forall o_{\nu\ell} \in act_{i-1}(o_\nu).\ o_{\mu k} <_{i-1} o_{\nu\ell}$$

by definition (4) and

$$o_{\mu k} <_{i-1} o_{\nu\ell} \Leftrightarrow \forall o_{\mu k\varrho} \in act_0(o_{\mu k}).\forall o_{\nu\ell\sigma} \in act_0(o_{\nu\ell}).\ o_{\mu k\varrho} <_0 o_{\nu\ell\sigma}$$

by the induction hypothesis. Taking both equivalences together, the claimed statement (5) follows from the definition (3) of act_0.

For the proof of (ii) we also apply induction on i, the case $i = 0$ being captured by Definition 3. For $i > 0$ and $o_{j\mu} <_i^{(j)} o_{j\nu}$ the well-definedness condition (1) implies $o_{j\mu k} <_{i-1}^{(j)} o_{j\nu\ell}$ for all $o_{j\mu k} \in act(o_{j\mu})$, $o_{j\nu\ell} \in act(o_{j\nu})$. By induction hypothesis we get $o_{j\mu k} <_{i-1} o_{j\nu\ell}$. Hence, the claimed result $o_{j\mu} <_i o_{j\nu}$ follows from the definition of $<_i$ in (4). □

2.3 Partial Schedules

The notion of n-level-schedule describes the interleaved execution of n-level-transactions. Temporal precedence on level L_i is expressed by the partial order $<_i$. Since transactions are built at run-time, we are also interested in *partial schedules*, where some of the later operations are omitted. These will be composed from *n-level-prefixes* of transactions in the same way, as (complete) schedules are composed from transactions.

Definition 4 Let T_j be an n-level-transaction. An *(n-level-)prefix* of T_j consists of subsets $\mathfrak{P}_i^{(j)} \subseteq \mathfrak{O}_i^{(j)}$ $(i = 0, \ldots, n)$ such that

- $o_{j\alpha} <_i^{(j)} o_{j\beta} \wedge o_{j\beta} \in \mathfrak{P}_i^{(j)} \Rightarrow o_{j\alpha} \in \mathfrak{P}_i^{(j)}$ and

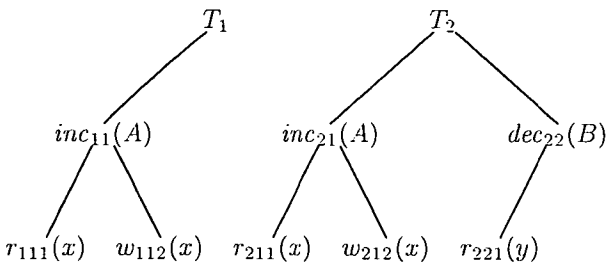

Figure 3: Partial 2-level-schedule

- $o_{j\alpha} \in \mathfrak{P}_i^{(j)} \Rightarrow trans(o_{j\alpha}) \in \mathfrak{P}_{i+1}^{(j)}$

hold, whenever the involved operations are defined. □

Formally, a prefix is different from a transaction unless we have $\mathfrak{P}_i^{(j)} = \mathfrak{O}_i^{(j)}$ for all i. On the other hand the selection of subsets for a prefix also defines an underlying subtree of the index-tree. Therefore, we may treat prefixes, as if they were (complete) transactions. In particular, we also have precedence relations $<_i^{(j)}$ on prefixes which result from restricting the corresponding relations associated with the transaction.

Furthermore, we may define schedules on the basis of prefixes using Definition 3. In this case we talk of a *partial n-level-schedule* and write $(\mathfrak{P}_n, \ldots, \mathfrak{P}_0, <_0)$ for this. Here $\mathfrak{P}_n = \{P_1, \ldots, P_k\}$ is a set of n-level-prefixes, $\mathfrak{P}_i = \bigcup_{j=1}^{k} \mathfrak{P}_i^{(j)}$ and $<_0$ is a partial order on $\mathfrak{P}_0$ containing all $L_0^{(j)}$-precedence relations restricted to $\mathfrak{P}_0$.

If all P_j are transactions, i.e. $P_j = T_j$, then we talk of a *complete* schedule.

Example 5 Figure 3 shows a partial schedule, where the tree rooted at T_j is a prefix of the 2-level-transaction T_j in Figure 1 ($j = 1, 2$). □

It is easy to see that each partial schedule can always be extended to a complete schedule by simply extending $<_0$ in some way compatible with the required extension of the L_0-precedence relations.

Conversely, given a complete schedule $(\mathfrak{O}_n, \ldots, \mathfrak{O}_0, <_0)$, we may choose a subset $\mathfrak{P}_0 \subseteq \mathfrak{O}_0$ such that $o_\alpha <_0 o_\beta$ with $o_\beta \in \mathfrak{P}_0$ implies $o_\alpha \in \mathfrak{P}_0$. Then $\mathfrak{P}_0$ induces a canonical partial schedule $(\mathfrak{P}_n, \ldots, \mathfrak{P}_0, <_0 |_{\mathfrak{P}_0})$. Such a partial schedule will be called a *prefix* of the given complete n-level-schedule. In this way partial schedules describe the interleaving of transactions in progress.

2.4 Conflict Serializability

The basic idea of multi-level concurrency control is to use the semantics of operations in level-specific, symmetric *conflict relations* $CON_i \subseteq \mathcal{O}_i \times \mathcal{O}_i$. Non-conflicting

operations should commute. In particular, it is natural to assume that conflicts can only occur on the same object, i.e. $((op_1, x), (op_2, y)) \in CON_i \Rightarrow x = y$.

Same as with precedence relations the intention behind the conflict relations forces us to require the following *conformity condition*: If $(o_\mu, o_\nu) \in CON_i$ holds for some $o_\mu, o_\nu \in \mathfrak{O}_i$, then there should exist $o_{\mu k} \in act(o_\mu)$ and $o_{\nu\ell} \in act(o_\nu)$ with $(o_{\mu k}, o_{\nu\ell}) \in CON_{i-1}$. The fundamental idea of multi-level transactions is that there may be low-level conflicts that do not stem from higher-level conflicts. Thus, the opposite of this condition need not to hold. In the sequel we shall tacitly assume that the conformity condition is satisfied by all schedules.

Example 6 Increments and decrements commute with one another. Therefore, for the transactions in Figure 1 we would like to use

$$((op_1, x), (op_2, y)) \in CON_1 \Leftrightarrow (op_1 = upd \vee op_2 = upd) \wedge x = y$$

assuming $\mathfrak{F}_1 = \{inc, dec, upd\}$. Analogously,

$$((op_1, x), (op_2, y)) \in CON_0 \Leftrightarrow (op_1 = w \vee op_2 = w) \wedge x = y$$

assuming $\mathfrak{F}_0 = \{r, w\}$. Note that the L_0-conflict relation is the usual one used for flat transactions.

Intuitively, the schedule in Figure 1 seems to be acceptable, but the level-by-level schedule $S_{2,0}$ in Figure 2 is not. The reason is that by omitting the L_1-operations we lost the information that the schedule is equivalent to the sequence $T_1; T_2$. Otherwise said, there are no conflicts on level L_1. Thus, multi-level transactions may be expected to increase concurrency, which will be made explicit in the following. □

We have to extend the notion of conflict-serializability to multi-level transactions to make these arguments rigorous. First, an n-level-schedule with a total order $<_n$ is called *serial*. Then *serializability* means equivalence to a serial schedule in the following formal sense.

Definition 5 Let $(\mathfrak{O}_n, \mathfrak{O}_{n-1}, \ldots, \mathfrak{O}_0, <_0)$ be an n-level-schedule with induced partial orders $<_i$ on level i. Let CON_i $(i = 0, \ldots, n-1)$ be conflict relations. Define

$$o_{j\mu} \rightarrow_i o_{j'\nu} \Leftrightarrow j \neq j' \wedge (o_{j\mu}, o_{j'\nu}) \in CON_i \wedge o_{j\mu} <_i o_{j'\nu} \tag{6}$$

for $o_\mu, o_\nu \in \mathfrak{O}_i$.

Then two n-level-schedules are called *(conflict-)equivalent* iff their associated relations $\rightarrow_i$ coincide for all $i = 0, \ldots, n-1$. An n-level schedule which is conflict-equivalent to a serial one, is called *(n-level-)serializable*. □

From the early studies of multi-level transactions [1, 24] it is well known that n-level-serializability can be detected from the level-by-level schedules $S_{i,i-1}$.

Lemma 2 *An n-level-schedule S is n-level-serializable iff all its level-by-level schedules $S_{i,i-1}$ $(0 < i \leq n)$ are serializable.* □

It is opportune to add a remark on partial schedules here. We shall call a partial schedule *serializable* iff it can be extended to a complete serializable schedule.

Example 7 Using the conflict relations from Example 6 it is easily verified that the schedule in Figure 1 is conflict serializable, whereas the one in Figure 2 is not. This was already stated above.

The partial schedule in Figure 3 can be extended to the one in Figure 1, hence is also serializable. □

Note that transactions in a serial schedule may leave the system and need not be considered any more. Serializability implies that transactions – not prefixes – may leave the system, if they can be brought into the first position in an equivalent serial schedule.

2.5 Recoverable Schedules

One desirable property of schedules for the flat transaction model was *recoverability*. Informally, this means that committed transactions should never be rolled back later. This can be expressed by the fact that if a transaction T_j reads from another transaction T_i, i.e. $w_{ik_1}(x) \rightarrow_0 r_{jk_2}(x)$ holds for some L_0-object x and suitable indices k_1, k_2, then whenever T_j commits, T_i must do so, too. In order to guarantee this property the commit of T_i must occur before the commit of T_j.

In order to generalize these notions to multi-level transactions, we first consider the read-from-relation. $w_{ik_1}(x) \rightarrow_0 r_{jk_2}(x)$ represents a strong conflict in the sense that an *abort-dependency* is implied: if T_i aborts, then T_j must do so, too. It is not sufficient to consider just the associated relations $\rightarrow_i$. For example, we could also have $r_{ik_1}(x) \rightarrow_0 w_{jk_2}(x)$ without abort-dependency. Hence, T_j may commit before T_i. If accidentally T_i aborts later on, this will not influence T_j anymore.

The difference between these two situations cannot be explained without regarding the "effects" of the operations. Roughly spoken, an object x on any level L_i has a value, say $\sigma(x)$ before the execution of an L_i-operation $op_\alpha(x)$ and a value $\tau(x)$ after that execution. The effect of the operation can therefore be expressed by the set $\{-\sigma(x), +\tau(x)\}$ or by $\emptyset$ in the case we have $\sigma(x) = \tau(x)$.

Now note that in our motivating example $w_{ik_1}(x) \rightarrow_0 r_{jk_2}(x)$ for L_0-operations the effect of the sequence $w_{ik_1}(x); r_{jk_2}(x)$ differs from the effect of $r_{jk_2}(x)$, whereas for $r_{ik_1}(x) \rightarrow_0 w_{jk_2}(x)$ the effects of the sequence $r_{ik_1}(x); w_{jk_2}(x)$ and of $w_{jk_2}(x)$ coincide. We now take this observation as a cornerstone for the generalization of recoverability on level L_i.

Definition 6 Let $\mathfrak{L} = ((\mathfrak{D}_{n-1}, \mathfrak{F}_{n-1}), \ldots, (\mathfrak{D}_0, \mathfrak{F}_0))$ be an *n-level-system* and assume sets V_i of values for each level L_i $(i = 0, \ldots, n-1)$. A *state* of an L_i-object $x \in \mathfrak{D}_i$ is an element $\sigma(x) \in V_i$. An *effect* on an L_i-object $x \in \mathfrak{D}_i$ is either a set $\{-\sigma(x), \tau(x)\}$, where $\sigma(x)$ and $\tau(x)$ are different states of x, or $\emptyset$. □

Now, we may assume that each L_i-operation $op_\alpha(x)$ – more generally: each sequence of L_i-operations on the same object x – has an effect on x. Of course, this effect depends on the content of the database. With these initial remarks we can now generalize the read-from-relation.

Definition 7 Let T_j, $T_{j'}$ be two n-level-transactions ($j \neq j'$) and $o_{j\mu}(x)$, $o_{j'\nu}(x)$ be two of their L_i-operations. We say that $o_{j'\nu}(x)$ *strongly depends* on $o_{j\mu}(x)$ (notation: $o_{j\mu}(x) \twoheadrightarrow_i o_{j'\nu}(x)$) iff $o_{j\mu}(x) \rightarrow_i o_{j'\nu}(x)$ holds and the effect of the sequence $o_{j\mu}(x); o_{j'\nu}(x)$ differs from the effect of $o_{j'\nu}(x)$. □

Note that for the flat transaction model the chosen definition turns only write-read-conflicts into strong dependencies – as desired.

Example 8 Consider L_1-operators *upd* for update, *inc* and *dec* for increment and decrement and a read-only operator *fetch*. Then again, we have $upd_{ik_1}(A) \twoheadrightarrow_1 fetch_{jk_2}(A)$, but $fetch_{ik_1}(A) \not\twoheadrightarrow_1 inc_{jk_2}(A)$. □

The second task is to generalize the abort-dependency resulting from $o_{j\mu}(x) \twoheadrightarrow_i o_{j'\nu}(x)$. For this we may assume that each operation in a partial schedule may abort or commit. This can be expressed by marking the operations in a partial schedule by c or a, respectively. Let $m(o)$ be the marking of the operation o. If we consider transactions in progress, it may happen that some operation which implements o has not yet been committed nor aborted. In this case we cannot assign a mark to o, which turns a marking m into a partial mapping.

Furthermore, all operations that implement an operation o, i.e. all operations $o' \in act(o)$, must commit before o can commit. Formally, this can be expressed by $m(o) = c \Rightarrow m(o') = c$. Analogously, all operations o' that must preced o, expressed by the precedence relation $o' <_i^{(j)} o$, must commit before o. This leads to the following definition.

Definition 8 Let $S = (\mathfrak{P}_n, \ldots, \mathfrak{P}_0, <_0)$ be a partial schedule. A *marking* of S is a partial mapping $m : \bigcup_{i=0}^{n} \mathfrak{P}_i \nrightarrow \{c, a\}$ such that the following holds:

1. If $(o) \subseteq \mathfrak{P}_{i-1}$ holds for $o \in \mathfrak{P}_i$, then $m(o)$ must be defined.
2. Whenever $m(o) = c$ and $o' \in act(o)$ hold, $m(o')$ is also defined with $m(o') = c$. Whenever $m(o) = a$ holds, there must exist some $o' \in act(o)$ with $m(o') = a$.
3. Whenever $o' <_i o$ holds, then $m(o') = c$ must hold.

A pair (S, m) with a partial schedule S and a marking m of S will be called a *marked schedule*. □

The first condition simply restricts attention to marked schedules, in which all operations are marked if they can be marked. The second condition expresses the requirement that all operations that implement a committed operation must have

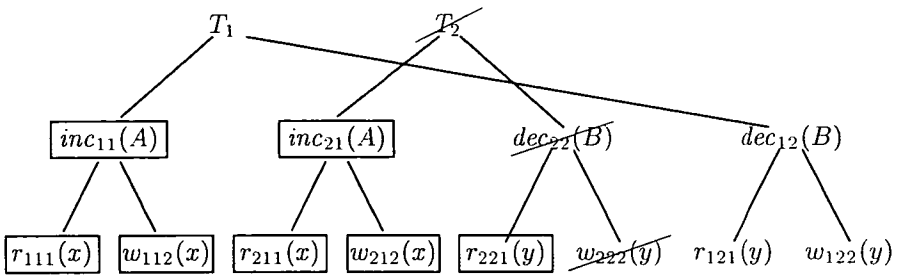

Figure 4: Marked multi-level schedule

been committed, too. Secondly, this condition expresses the analogue that among the operations that implement an aborted operation there must be at least one which has been aborted, too. The third condition expresses that if an operation has been completed before another one, it must have committed.

Example 9 Consider the marked schedule in Figure 4 with a boxed entry marking a committed operation and a crossed out operation marking an aborted one. The underlying schedule is the one from Figure 1. □

The notion of a marked schedule now allows recoverability to be generalized to multi-level schedules.

Definition 9 A schedule $(\mathfrak{O}_n, \ldots, \mathfrak{O}_0, <_0)$ is called *recoverable on level* L_i iff for all prefixes S, all markings m of S and all $j \neq j'$

$$o_{j\mu k} \twoheadrightarrow_{i-1} o_{j'\nu\ell} \wedge m(o_{j'\nu}) = c \Rightarrow m(o_{j\mu}) = c$$

holds. □

Note that in contrast to recoverability for flat schedules recoverability on level L_i does not completely exclude committed operations from being rolled back later. However, the abort of a committed L_i-operation will only be triggered by the abort of $trans(o_\mu)$. We discuss recoverability together with the protocols presented in Sections 3 and 4.

Finally, we may also generalize the stronger notions of *cascade-freeness* and *strictness* to multi-level schedules.

Definition 10 Let $S = (\mathfrak{O}_n, \ldots, \mathfrak{O}_0, <_0)$ be an n-level-schedule.

1. S is called *cascade-free on level* L_i iff for all prefixes S' of S, all markings m of S' and all $j \neq j'$ it is true that whenever $o_{j\mu k} \twoheadrightarrow_{i-1} o_{j'\nu\ell}$ holds, then $m(o_{j\mu})$ must be defined.

2. S is called *strict on level* L_i iff for all prefixes S' of S, all markings m of S' and all $j \neq j'$ it is true that whenever $o_{j\mu k} \rightarrow_{i-1} o_{j'\nu\ell}$ holds, then $m(o_{j\mu})$ must be defined. □

We shall discuss cascade-freeness and strictness together with the protocols presented in the next two sections. As a first result which is obvious from the definitions we notice that strictness implies cascade-freeness.

Example 10 Consider the schedule from Figure 1 with a total order $<_0$. Then we have the strong dependencies $w_{112}(x) \twoheadrightarrow_0 r_{211}(x)$ and $w_{222}(y) \twoheadrightarrow_0 r_{121}(y)$ on level L_0 and no such dependencies on level L_1. Obviously, in the marked schedule in Figure 4 the conditions for strictness, cascade-freeness and recoverability are satisfied for level L_1.

More generally, we can show that the schedule from Figure 1 is indeed recoverable on level L_1. If $w_{112}(x)$, $r_{211}(x)$ and $inc_{21}(A)$ with $m(inc_{21}(A)) = c$ (or $w_{222}(y)$, $r_{121}(y)$ and $dec_{12}(B)$ with $m(dec_{12}(B)) = c$, respectively) occur in a marked prefix, then $m(inc_{11}(A)) = c$ (or $m(dec_{22}(B)) = c$, respectively) must hold by the third condition in Definition 8.

We can also show that the schedule is cascade-free on level L_1. If we consider a prefix, in which $w_{112}(x)$ (or $w_{222}(y)$, respectively) occurs, then by the first condition in Definition 8 $m(inc_{11}(A))$ (or $m(dec_{22}(B))$, respectively) must be defined.

The same argument applies, if we consider $\rightarrow_0$, which gives

$$r_{111}(x) \rightarrow_0 w_{212}(x),\ w_{112}(x) \rightarrow_0 r_{211}(x),\ w_{112}(x) \rightarrow_0 w_{212}(x)$$

and

$$r_{221}(y) \rightarrow_0 w_{122}(y),\ w_{222}(y) \rightarrow_0 w_{121}(y),\ w_{222}(y) \rightarrow_0 w_{122}(y)\ .$$

This shows that the schedule is even strict on level L_1. □

3 Locking Protocols

Locking protocols for multi-level transactions have been investigated from the very beginning [24]. Therefore, we shall only describe very briefly the gist of these protocols.

According to our assumption that only those operations give rise to conflicts, which access the same object, it is sufficient to concentrate on the operators. Thus, for each L_i-operator $op \in \mathfrak{F}_i$ we define a specific lock $lock_{op}$. Then, each L_i-operation $op_{\mu k}(x)$ may only be executed after setting a lock, namely $lock_{op}$, on the object x. In addition we associate with this lock the index μ of the issuing operation $o_\mu = trans(o_{\mu k})$. After its commit, o_μ must release all its locks.

Same as with read-locks for flat transactions, an L_i-object x may hold several locks at a time, provided the associated operations do not conflict with each other.

Definition 11 Let $lock_{op_1}$ and $lock_{op_2}$ be locks on object $x \in \mathfrak{D}_i$ issued by the L_i-operations $o_{\mu k}$ and $o_{\nu \ell}$, respectively. These locks are called *incompatible* iff $o_{\mu k} \rightarrow_i o_{\nu \ell}$ or $o_{\nu \ell} \rightarrow_i o_{\mu k}$ holds. □

Thus, an operation may only set a lock on x, if this is not incompatible with any existing lock on x. Otherwise, the operation has to be aborted or must wait until all incompatible locks on x are released.

This basic idea underlying multi-level locking protocols can be extended in the usual way to define *two-phase locking* (2PL) as well as conservative or strict variants. In 2PL we have a *growing phase*, in which all locks are acquired, but none can be released, followed by a *shrinking phase* in which existing locks will be released, but no new lock can be acquired. In conservative 2PL (con-2PL) all locks are set before the operation actually starts. In strict 2PL (str-2PL) no lock will be released before commit or abort.

Example 11 Consider the schedule in Figure 1 assuming a total order $<_0$ from left to right. On level L_0 we have the usual read- and write-locks, i.e. $lock_r$ and $lock_w$ using our current notation. Only two read-locks are compatible with each other. Thus, all locks on L_0-objects can be set and released by 2PL without any problems.

On level L_1 we have locks $lock_{inc}$, $lock_{dec}$ and $lock_{upd}$ for the increment-, decrement- and general update-operation. Only the update-lock is incompatible to all other locks. Then, also all locks on L_1-objects can be set and released by 2PL. Hence, the schedule will be accepted by 2PL. □

The example indicates that schedules accepted by 2PL will be serializable. Such a result stating the correctness of 2PL for multi-level schedules is well-known from the early literature [24].

Theorem 1 *A multi-level-schedule accepted by the use of 2PL on each level is always serializable.*

Proof. Suppose we have $o_{j\mu k} \rightarrow_i o_{j'\nu\ell}$ for $j \neq j'$ on level L_i. The conformity assumption for conflict relations implies $(o_{j\mu}, o_{j'\nu}) \in CON_{i+1}$. The incompatibility of the corresponding locks and the 2PL-strategy to keep the first of these locks until $o_{j\mu}$ has committed implies $o_{j\mu} <_{i+1} o_{j'\nu}$.

Taken together, we obtain $o_{j\mu} \rightarrow_{i+1} o_{j'\nu}$ and by induction $T_j <_n T_{j'}$.

If 2PL accepted a non-serializable schedule, we would also have $o_{j'\mu'k'} \rightarrow_{i'} o_{j\nu'\ell'}$ on some level $L_{i'}$. Hence, $T_{j'} <_n T_j$ holds, too, which is impossible for a partial order. □

Example 12 Now consider the schedule in Figure 5. Taking the same locks and incompatibility relations as before, T_2 will not be able to set the update-lock on object A before the commit of T_1, because T_1 holds an incompatible increment-lock on A. This implies that the shown interleaving in Figure 5 is not acceptable by 2PL.

Nevertheless, the shown schedule is serializable, which demonstrates that the converse of Theorem 1 does not hold. □

As a straightforward result we show that strict 2PL leads to recoverable and strict schedules.

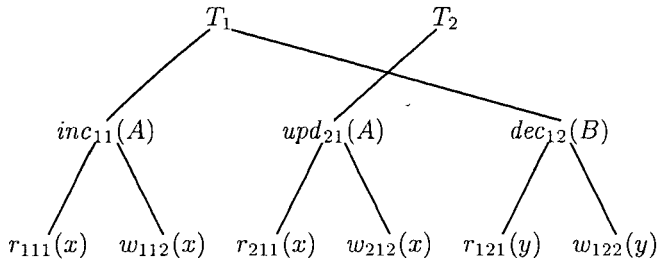

Figure 5: Serializable schedule, not acceptable by 2PL

Proposition 1 *If strict 2PL is used on level L_i, then the resulting schedule is recoverable and strict on level L_i.*

Proof. Assume $o_{j\mu k}(x) \twoheadrightarrow_{i-1} o_{j'\nu\ell}(x)$ and $m(o_{j'\nu}) = c$. Then $o_{j'\nu}$ must have acquired a lock on x, which is only possible in the shrinking phase of $o_{j\mu}$. According to the definition of str-2PL this happens after the commit of $o_{j\mu}$. Hence $m(o_{j\mu}) = c$ holds, i.e. the schedule is recoverable on level L_i.

Next assume $o_{j\mu k}(x) \rightarrow_{i-1} o_{j'\nu\ell}(x)$. According to the definition of str-2PL, $o_{j'\nu}$ can only appear in marked schedules with $m(o_{j\mu})$ being defined. Hence the schedule is strict on level L_i. □

4 A Hybrid Concurrency Control Protocol

We now present the FoPL (**F**orward oriented Concurrency Control with **P**reordered **L**ocking) protocol, which ensures serializability by exploiting the level-by-level schedules $S_{i,i-1}$. Then we shall discuss its correctness and completeness with respect to serializability and the issues of recoverability and strictness.

4.1 The Basic FoPL Protocol

The basic structure follows the idea of optimistic protocols or hybrid protocols such as ODL [11]. Thus, FoPL consists of three phases: the propagation, validation and commit-phase. In the *propagation-phase* the operations at the various levels L_i are executed. In addition, some kind of control-structure consisting *flaglists* for the objects and *access-lists* for the operations is built up and will be used later to decide, whether on operation commits or aborts.

The task of the *validation-phase* is to perform this decision. The flaglists are used to detect, whether the interleaved execution of the operations has lead to a situation that forces an abort or not. Finally, in the *commit-phase* the commit or abort is executed. We shall see that the commit-case is the easier one: if in-place updates are used, then the only task is to remove flags from flaglists. The abort-case requires additional efforts for rollback. This will be postponed to Section 7 on recovery.

4.1.1 Propagation

In the propagation phase the operations of a schedule are executed according to some order $<$ which extends $<_0$. In practice this order is built dynamically according to the invocation of transactions. In centralized systems $<$ may be assumed to be total, but in distributed systems this is not necessary. In contrast to other optimistic or hybrid protocols changes to the database are made persistent immediately. We shall also discuss what happens, if changes are only stored in private buffers made persistent in the commit phase if at all.

Since an L_i-operation o_μ is implemented by $act(o_\mu)$, we mark the objects in $\mathfrak{D}_{i-1}$ that are accessed by o_μ. If $o_{\mu k} \in act(o_\mu)$ is the operation (op, A), then we use the *flag* (op, μ) on A. We use a *flaglist* ZL_A for each object $A \in \mathfrak{D}_{i-1}$ (and each $i = 1, \ldots, n$), which is built dynamically extending $<_{i-1}$.

In addition, we use *access lists* $AS_i^{(\mu)}$ to keep track of the objects accessed by o_μ. In order to see not only the accessed objects but also the way they are accessed, we take $AS_i^{(\mu)} = act_{i-1}(o_\mu)$, i.e. we use the implementing operations.

Example 13 In Figure 1 the flaglists ZL_A and ZL_B on level L_1 are constructed as $ZL_A = inc_1 inc_2$ and $ZL_B = dec_2 dec_1$. □

When appending a flag to a flaglist an exclusive *short-term-lock* on the flaglist is used. This guarantees that the append-operation is atomic. In particular, concurrent access to the same flaglist can be executed without the risk to loose flags. Deadlocks are not possible, because an operation holds only one lock at a time. Flags will be removed again from flaglists during the commit-phase.

In addition, we may assume that setting the flag is executed before the execution of the operation. For L_0-operations it is necessary to keep this short-term-lock until the operation itself is finished, because this guarantees that there is is no undesired interference with other L_0-operations.

4.1.2 Validation

If all operations in $act(o_\mu)$ have been executed, o_μ initiates its validation. For this, FoPL has to test if all flags that stem from $act(o_\mu)$ are still set. As we shall see below in the paragraph on the commit-phase, flags may have been discarded from a flaglist by another operation.

For the flaglists of all objects $A \in \mathfrak{D}_{i-1}$, which were accessed by $act(o_\mu)$ during the propagation phase, exclusive locks will be requested and kept until the end of the commit-phase. To avoid deadlocks the locks are requested in a total order, which justifies the naming of the protocol. It is not necessary to request locks on the L_{i-1}-objects themselves, since only the flaglists are analyzed. In Section 5 we shall discuss an alternative strategy, which dispenses completely with locks.

The involved objects can be recognized from the access list $AS_i^{(\mu)}$. In particular, $AS_i^{(\mu)}$ indicates all the flags that should still be set.

If at least one flag is missing, the operation o_μ must abort. Otherwise, FoPL tests, whether o_μ was *successful*. This is the case, if none of the objects in $\mathfrak{D}_{i-1}$

accessed by o_μ was accessed by some other operation o_ν before. This can be detected from the flaglists.

Definition 12 An L_i-operation o_μ is *blocked* on an object $A \in \mathfrak{D}_{i-1}$ iff there are flags (op_1, ν) and (op_2, μ) in ZL_A with $\nu \neq \mu$ such that (op_1, ν) precedes (op_2, μ) and $((op_1, A), (op_2, A)) \in CON_{i-1}$ holds.

An L_i-operation o_μ is *successful on an object* $A \in \mathfrak{D}_{i-1}$ iff it is not blocked on A. An L_i-operation o_μ is *successful* iff it is successful on all objects accessed by $act(o_\mu)$. □

If the operation o_μ is successful, it can commit, otherwise it must abort. Both *actions* (commit/abort) are accomplished during the commit phase.

4.1.3 Commit

If an L_i-operation o_μ may commit, the flaglists of all objects $A \in \mathfrak{D}_{i-1}$, which were accessed by $act(o_\mu)$ during the propagation phase, have to be updated. For this the locks requested in the validation-phase are kept. Then all flags from $act(o_\mu)$ have to be removed. After removing the flags, the locks will be released thereby terminating the commit-phase.

If an L_i-operation o_μ must abort, all operations in $act(o_\mu)$ must abort. In this case the flags from $o_{\mu k}$ may still be set or not. In the first case, a compensation is executed, if possible. If not, the object updated by $o_{\mu k}$ has to be replaced by its *before image*. Finally, all remaining flags and all dependent flags have to be deleted.

Definition 13 A flag z from o_μ *depends* on another flag z' from o_ν, iff z' precedes z in ZL_A and $(o_\nu, o_\mu) \in CON_i$ holds or z depends on z'' and z'' depends on z' for some flag z''. □

If a compensation operation $o_{\mu k}^{-1}$ is initiated to abort $o_{\mu k}$, it must be applied to the before image of the *first* operation $o_{\nu l}$, whose flag depends on the flag of $o_{\mu k}$. Because all operations which depend on $o_{\mu k}$ have to abort later on, it is also possible to abort those operations before aborting $o_{\mu k}$. Therefore, a rollback recovery can be invoked.

In the second case there is nothing to do, because an earlier abort from another operation has overwritten the update from $o_{\mu k}$ or the operation was already aborted by the rollback-recovery.

4.2 Lazy Aborts: The FoPL$^+$ Protocol

In order to minimize the number of aborts we may employ the alternative to force an operation to wait and to restart after some time period. We call this *lazy abort*. If FoPL is combined with lazy-abort, the resulting protocol is called FoPL$^+$.

Since conflicts on higher levels are assumed to occur not too often, we may hope that the preceding conflicting flag has been deleted in the meantime. Thus, aborts will only occur, if they are really unavoidable.

As a disadvantage note that deadlocks may occur, if the (transitive closure of the) waiting-for-relation contains a cycle, e.g. an operation o_μ waits for o_ν and o_ν waits for o_μ. In this case the easiest solution is to abort both operations, because o_ν has read from o_μ and o_μ has read from o_ν. We shall discuss alternatives in the next section. Thus, phantom deadlocks cannot occur. If a deadlock is detected, it can be resolved by deleting one flag, which is involved in the deadlock. Deadlocks can be detected with known techniques [5, 18].

Note, however, that a waiting operation does not prevent any object from being accessed. Thus, the possibility of deadlocks is less critical compared with lock protocols.

Example 14 First consider the schedule in Figure 1. Then the progress of the flaglists on L_0-objects x, y and L_1-objects A, B is as follows:

	1	2	3	4	5	6	
ZL_A	inc_1	inc_1	inc_1 inc_2	inc_1 inc_2	inc_1 inc_2	inc_1 inc_2	
ZL_B					dec_2	dec_2	
ZL_x	r_{11}	r_{11} w_{11}	r_{21}	r_{21} w_{21}			
ZL_y					r_{22}	r_{22} w_{22}	

	7	8	9	10	
	inc_1	inc_1	inc_1		ZL_A
		dec_1	dec_1		ZL_B
					ZL_x
		r_{12}	r_{12} w_{12}		ZL_y

Here we assume that the commit of $inc_{11}(A)$ occurs between columns 2 and 3, the commit of $inc_{21}(A)$ between columns 4 and 5, the commit of $dec_{22}(B)$ and T_2 occur between columns 6 and 7, and finally, the commits of $dec_{12}(B)$ and T_1 occur between columns 9 and 10. Thus, the schedule will be accepted. □

Example 15 Consider the schedule in Figure 5, which was not acceptable for 2PL. Looking only at flaglists on L_1-objects we obtain (with FoPL$^+$):

	1	2	3	4
ZL_A	inc_1	inc_1 upd_2	inc_1 upd_2	upd_2
ZL_B			dec_1	

with T_1 committing between 3 and 4, T_2 committing after 4 and T_2 waiting from the beginning of 3 to the end of 4.

Thus, FoPL$^+$ will accept this schedule, but FoPL would abort T_2, since ZL_A cannot be permuted. □

4.3 Private Buffers

As an alternative to immediate in-place updates we may think of using private buffers as in other optimistic or hybrid protocols [11, 12]. Changes to the database objects are only stored in these private buffers during the propagation phase and made persistent in the commit phase if at all.

Since higher-level operations are by assumption implemented by lower-level operations, there is only a need for such buffers on the level L_0. Consequently, we may only expect changes to the protocol on that level.

The crucial point is now that results of operations affecting the database are not visible to other L_0-operations, as long as the corresponding L_1-operation has not finished its commit phase. Hence, after a successful commit of an L_1-operation, all its flags and all dependent flags on level L_0 have to be deleted. Furthermore, since the actual changes to the object are only performed at commit-time, it is insufficient to lock only flaglists. The referred objects must be locked, too. On the other side, the deletion of dependent flags is no longer necessary in the abort case.

Example 16 Consider the development in Example 14, but now assume assume that $inc_{11}(A)$ validates and commits after $r_{211}(x)$ has been performed. In this case the flag r_{21} in ZL_x will be removed causing the later abort of $inc_{21}(A)$. This is correct, since otherwise a wrong value might be used by $inc_{21}(A)$ ("dirty read"), and the update by $inc_{11}(A)$ will get lost ("lost update"). □

Whether it is advantageous to apply FoPL with in-place updates or private buffers on level L_0 depends on the probability of conflicts occurring on level L_0. Note that it is even possible to mix both strategies, i.e. to let some transactions – or L_1-operations – use private L_0-buffers, whereas others use in-place updates. As a rule of thumb, if it can be expected that L_1-operations will commit, then choose in-place updates, because this will trigger less rollbacks.

4.4 Correctness and Completeness

Let us now investigate the correctness and completeness of the FoPL protocol with respect to serializability. In order to distinguish between the basic FoPL protocol and the optimization through lazy aborts we use FoPL$^+$ to indicate the enhanced protocol.

Theorem 2 *Every n-level-schedule accepted by the FoPL protocol is n-level-serializable.*

Proof. If a transaction T_j commits, this also applies to all operations o_μ defining it – at different levels. This is only possible, if all these operations are successful on all objects. These implies that $o_\nu \not\rightarrow_i o_\mu$ holds for all other operations o_ν issued by different transactions, and the schedule is conflict-equivalent to a serial schedule with first transaction T_j.

Proceeding inductively and exploiting the fact that flags from submitted transactions will be removed, we obtain an equivalent serial schedule. □

Obviously, since commit for FoPL$^+$ works in the same way as for FoPL, the correctness result carries over to the optimized version with lazy abort.

Corollary 1 *Every n-level-schedule accepted by the FoPL$^+$ protocol is n-level-serializable.* □

In addition to this correctness result, we may also obtain a completeness result for FoPL$^+$, i.e. if we adopt the alternative waiting strategy discussed above. The central argument of the proof states that a deadlock in the waiting graph may only occur iff $o_\mu \to_i o_\nu$ and $o_\nu \to_i o_\mu$ both hold. But this means that the schedule is not serializable.

Lemma 3 *A deadlock in FoPL$^+$ occurs iff the corresponding partial schedule is not serializable.*

Proof. Suppose we have a deadlock between L_i-operations indicated by flaglists $ZL_A = o_\mu p_\nu$ and $ZL_B = q_\nu s_\mu$. For the corresponding L_{i-1}-operations we obtain $o_{\mu k}(A) \to_{i-1} p_{\nu\ell}(A)$ and $q_{\nu m}(B) \to_{i-1} s_{\mu n}(B)$ for suitable indices k, ℓ, m, n. This states that the level-by-level schedule $S_{i,i-1}$ is not serializable.

Conversely, assume a non-serializable level-by-level schedule $S_{i,i-1}$, i.e. $o_{\mu k}(A) \to_{i-1} p_{\nu\ell}(A)$ and $q_{\nu m}(B) \to_{i-1} s_{\mu n}(B)$ holds for L_{i-1}-objects A, B, L_i-operations o_μ, o_ν and $o_{\mu k}(A), s_{\mu n}(B) \in act(o_\mu)$, $p_{\nu\ell}(A), q_{\nu m}(B) \in act(o_\nu)$. This implies the flaglist to contain $ZL_A = o_\mu p_\nu$ and $ZL_B = q_\nu s_\mu$. Since no permutation is possible in ZL_A nor ZL_B, o_μ waits for o_ν and vice versa. Hence, there is a deadlock. □

From this lemma and the preceding remarks the claimed completeness result follows immediately.

Theorem 3 *Every n-level-serializable schedule will be accepted by the FoPL$^+$ protocol.* □

4.5 Recoverability and Strictness

Finally, we investigate recoverability and strictness.

Proposition 2 *If FoPL (or FoPL$^+$) is used on level L_i, then the resulting schedule is recoverable.*

Proof. Assume $o_{j\mu k}(x) \twoheadrightarrow_{i-1} o_{j'\nu\ell}$ and $m(o_{j'\nu}) = c$. The first assumption implies that ZL_x contains $p_{j\mu} q_{j'\nu}$ with the corresponding L_{i-1}-operators p, q. The second assumption implies that $q_{j'\nu}$ could be removed from ZL_x. According to the definition of FoPL this is only possible, if $p_{j\mu}$ was removed earlier from ZL_x, i.e. $m(o_{j\mu})$ is defined.

If we had $m(o_{j\mu}) = a$, then the removal of $p_{j\mu}$ would have triggered the removal of the dependent flag $q_{j'\nu}$ which contradicts the fact that $o_{j'\nu}$ committed. □

However, in contrast to strict 2PL, FoPL (and FoPL$^+$) cannot guarantee strictness, not even cascade-freeness as can be seen from the next example. This is reflected

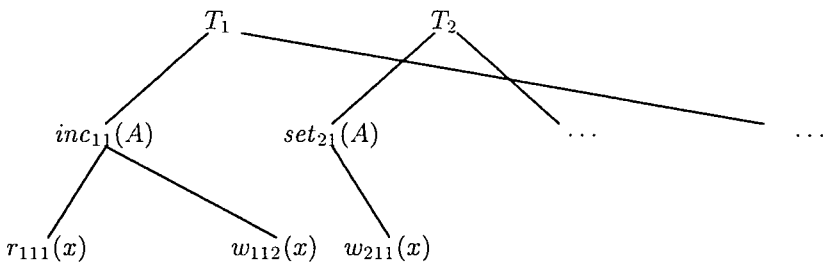

Figure 6: Partial FoPL schedule with cascade

in the protocol by the removal of dependent flags. Non-cascade-freeness is usual with optimistic or hybrid protocols. It is the price to be paid for the increase in transaction throughput resulting from the visibility of operation results before the final commit of a transaction.

Example 17 Consider the schedule sketched in Figure 6. Omitting the dotted parts we obtain a partial schedule with $inc_{11}(A) \twoheadrightarrow_1 set_{21}(A)$, but without $m(T_1)$ being defined. Hence, the schedule is not cascade-free on the top level L_2. □

5 Optimization of the Basic FoPL Protocol

We shall now discuss various optimizations of the basic FoPL protocol or the FoPL$^+$ protocol with lazy aborts. First we ask, whether the exclusive locks in the validation and commit phase are really needed. This will lead to the debatable noPL-strategy.

Next we shall handle rollbacks. The first optimization concerns the ability to detect necessary aborts before entering the validation phase. The second optimization discusses the use of partial rollbacks.

Finally, we consider the absorption of operations. If the effect of an operation does not depend on the execution of a preceding operation, this enables some rollbacks to be dispensed with or the enforcement of validation success.

5.1 Optimistic Locking

In principle, since validating operations only read flaglists, it is not necessary to lock these lists during validation. Furthermore, any other active operation may only add new flags at the end of the lists. Such new flags do not influence the validation result and consequently do not require locks either.

As a further optimization related to optimistic locking [21] it is not even necessary to keep the used exclusive locks during the whole commit-phase, but to release them immediately after changing the flaglist, since all other changes to the objects in question have been detected in the validation phase to commute. Thus, it is only necessary to guarantee the atomicity of the changes to the flaglists via short-term-locks. We shall talk of the noPL-strategy (**no** preordered locking).

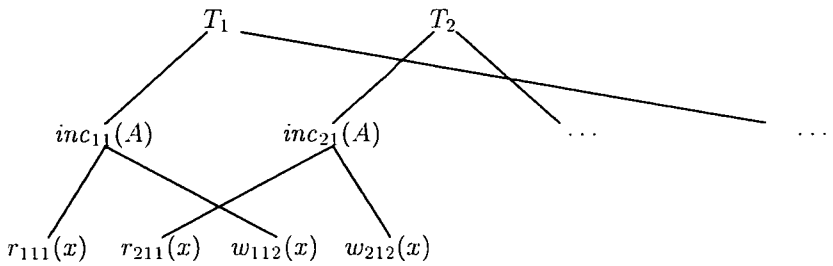

Figure 7: Deadlock in a FoPL$^+$ schedule

However, in the case of an abort such an early release of locks may lead to the removal of flags from operations that are uncritical otherwise. For example, it might not be possible to execute the operation at all due to the locked flaglist. In this case the noPL-strategy may lead to unnecessary aborts. The same applies for the commit, if private buffers are used.

On the other hand, with such an early release of locks we risk the removal of flags from operations that are uncritical otherwise, which may lead to unnecessary aborts.

Example 18 Consider the development in Example 14. Since all operations will commit, there will be no change, if we adopt the noPL-strategy.

However, things change, if we decide to abort $inc_{11}(A)$ and this decision is taken before the execution of $r_{211}(x)$. With the noPL-strategy flaglists are not locked, so it would be possible to execute $r_{211}(x)$ before changing the flaglist ZL_x. Then the flag r_{21} in ZL_x would be removed causing the later abort of $inc_{21}(A)$. Thus, with noPL we risk the unnecessary abort of T_2.

Similarly, consider the schedule in Example 15. With the noPL-strategy the commit of $inc_{11}(A)$ does not lead to a problem, but in the case of an abort the changes to the unlocked flaglist ZL_x may occur after $r_{211}(x)$. Then r_{21} will be deleted from ZL_x, which causes $upd_{21}(A)$ and T_2 to abort. □

It depends on the probability of concurrent access to the same object, whether the noPL-strategy is advisable or not.

5.2 Early and Partial Rollback

In the basic FoPL protocol the necessity to abort an operation and to trigger a rollback will be detected in the validation phase, if a corresponding flag is missing. As an alternative it is possible to inform an operation immediately, when one of its flags will be removed. This strategy of *early rollbacks* will probably prevent further operation from being executed, if we already know about their later abort.

It depends on the duration of operations, whether the communication overhead caused by early rollbacks is small compared with the time waste for operations to be aborted later. In general, early rollback may be advantageous on higher levels.

No matter, whether early rollbacks are applied or not, it is not necessary to rollback operations completely. Since only dependent flags are deleted, it is sufficient to do a partial rollback to the earliest time point, where none of these flags were set.

Partial rollbacks are also useful for removing deadlocks as seen in the next example.

Example 19 Consider the partial schedule in Figure 7, which leads to a deadlock with the flaglist $ZL_x = r_{11}r_{21}w_{12}w_{21}$. It is only necessary to partially rollback until we have $ZL_x = r_{11}$, and then to restart $w_{112}(x)$ again. In this case T_2 has been aborted, but not T_1. □

5.3 Absorption

Consider the case of a conflict $o_{j\mu}(x) \rightarrow_i o_{j'\nu}(x)$, where the second operation does not strongly depend on the first one. According to the definition of $\rightarrow_i$ this means that the second operation absorbs the first one.

Definition 14 Let $T_j, T_{j'}$ be two n-level-transactions ($j \neq j'$) and $o_{j\mu}(x)$, $o_{j'\nu}(x)$ be two of their L_i-operations. Then $o_{j'\nu}(x)$ *absorbs* $o_{j\mu}(x)$ (notation: $o_{j\mu}(x) \rightsquigarrow_i o_{j'\nu}(x)$) iff $o_{j\mu}(x) \rightarrow_i o_{j'\nu}(x) \wedge o_{j\mu}(x) \not\twoheadrightarrow_i o_{j'\nu}(x)$ holds. □

Absorption $o_{j\mu k}(x) \rightsquigarrow_{i-1} o_{j'\nu\ell}(x)$ allows a brute force strategy to be used when validating $o_{j'\nu}$. We simply remove the flag $p_{j\mu}$ set by $o_{j\mu k}(x)$ in ZL_x, if this makes $o_{j'\nu}$ successful on x. Of course, the deletion of $p_{j\mu}$ will cause $o_{j\mu}$ to be aborted later. Furthermore, we delete all dependent flags which stem from operations that strongly depend on $o_{j\mu k}(x)$.

This strategy immitates a schedule, where $o_{j\mu k}(x)$ was not executed. The strategy will be called *commit enforcement strategy*.

Example 20 As an alternative to the processing in Example 19 we could have used the commit enforcement strategy with $inc_{11}(A)$ which immediately gives $ZL_x = w_{21}$. This will also cause T_2 to abort, but without rolling back $w_{112}(x)$. □

6 Comparison of FoPL$^+$ with Locking

We start with a comparison of FoPL$^+$ with strict two-phase locking (str-2PL). As a probabilistic model for multi-level transactions is still missing, this discussion will necessarily remain preliminary. Nevertheless, we discuss both protocols with respect to implementation costs and transaction throughput.

6.1 Implementation Costs

FoPL$^+$ uses access lists $AS_i^{(\mu)}$ to keep track of the objects accessed by the L_i-level operation o_μ. If str-2PL is used, we must also keep track of the accessed objects to

be able to request and release locks. So, with respect to the costs of implementing these access-lists there is no difference between the protocols.

FoPL$^+$ uses flaglists for concurrency-control, and short-term-locks are always necessary when flaglists are accessed. Similarly, str-2PL must support a lock-table to keep track of the locks. This could be arranged as a list of locks for each object.

The first task for str-2PL is to check a locklist for conflicts each time a new lock is requested. This can be achieved by linear search. Even, if an L_i-operation o_μ already holds a lock on an L_{i-1}-object x requested by some $o_{\mu k}$, it is in general not possible to avoid conflict checking, when another operation $o_{\mu\ell}$ wants to access the same object x. Let us illustrate this by a simple example.

Example 21 Suppose $o_{j\alpha}$ holds a *fetch*-lock on the L_1-object A due to some operation $fetch_{j\alpha k}(A)$. Then it is possible, that another operation $o_{j'\beta}$ also holds a *fetch*-lock on A due to some operation $fetch_{j'\beta\ell}(A)$. The two *fetch*-locks are compatible to one another.

If $o_{j\alpha}$ now requires another lock on A, say an *inc*-lock due to $inc_{j\alpha k'}(A)$, this request must be rejected, as the required lock is incompatible with the *fetch*-lock held by $o_{j'\beta}$. □

On the other hand, FoPL$^+$ does not check anything on appending a flag to a flaglist. The check for conflicts is done in the validation-phase. For each operation $op_{j\alpha k}(x) \in act(o_{j\alpha})$ the first entries in the flaglist preceding $(op, j\alpha)$ have to checked for conflicts. This again leads to linear search.

Thus, for conflict-checking we may state that FoPL$^+$ produces an overhead over str-2PL: flaglists may be longer than locklists and they are accessed more frequently. However, this overhead seems not to be dramatic. In particular, the main parameter to validate this overhead is the number of different operations accessing the same object x within a short period of time. One major assumption for introducing multi-level transactions was that this number is rather small except for level L_0. So the only critical overhead could appear on level L_0, but here we usually have only short read-write sequences.

After commit, str-2PL has to access the lock-table again to release locks. This can be realized by linear searching the locklists associated with the relevant objects. FoPL$^+$ has to delete flags in the case of commit and abort. For abort – and also for L_1-commit, if private buffers are used – dependent flags have to removed either. For this there is no significant difference concerning the implementation costs of str-2PL and FoPL$^+$.

Finally, we must look at the implementation costs for deadlock detection. For this str-2PL has to implement a waiting graph on L_i-operations, which is updated each time a lock-request has been rejected and on commit and abort. The same applies to FoPL$^+$. In particular, the costs for deadlock detection are the same for both protocols. The major difference, however, is that with str-2PL locks are held on objects, whereas with FoPL$^+$ the operations on the waiting graph are independent from the execution of the transactions. This may have an impact on transaction throughput, as we shall discuss next.

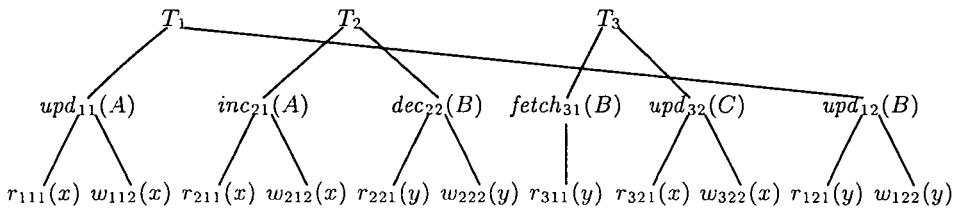

Figure 8: Non-serializable multi-level schedule with FoPL$^+$-overhead

As a first conclusion we may already state that there is no significant increase in implementation costs, when FoPL$^+$ is used instead of str-2PL. In any case, an increase in transaction throughput would justify the implementation overhead.

6.2 Transaction Troughput

As to transaction throughput the discussion will be based on three examples. The first one is Example 15, where we could show that FoPL$^+$ accepts the serializable schedule from Figure 5, whereas str-2PL would not. According to our completeness result (Theorem 3) for FoPL$^+$ this is no longer astonishing. With str-2PL the operation $upd_{21}(A)$ could first be started after the commit of T_1. This causes an overhead of one L_1-operation and two L_0-operation. In this case the advantages of FoPL$^+$ are evident.

Next we consider two other examples shown in Figures 8 and 9. The first of these examples shows an non-serializable schedule with an overhead for FoPL$^+$. The last example demonstrates the power of FoPL$^+$, when the optimizations with absorption on level L_0 and early rollbacks are employed.

Example 22 Consider the non-serializable schedule from Figure 8. In this case FoPL$^+$ would produce the following flaglists on L_1-objects:

ZL_A	upd_1 inc_2
ZL_B	dec_2 $fetch_3$ upd_1
ZL_C	upd_3

Then FoPL$^+$ would abort o_{22}, because it removes the shortest cycle in the waiting graph, and consequently also o_{31}, o_{32} and o_{12}. A restart – o_{22} must be restarted later than o_{31} and o_{12} may lead to the flaglists

ZL_A	upd_1 inc_2
ZL_B	$fetch_3$ upd_1 dec_2
ZL_C	upd_3

and all transactions would commit now. In this case four L_1-operations composed from seven L_0-operations must be repeated.

For str-2PL the schedule could not occur. Howeverr, due to the *upd*-lock on A held by T_1 the operation o_{21} can only start after T_1 has committed. Thus, the overhead of FoPL$^+$ consists only of two L_1-operations composed from three L_0-operations. □

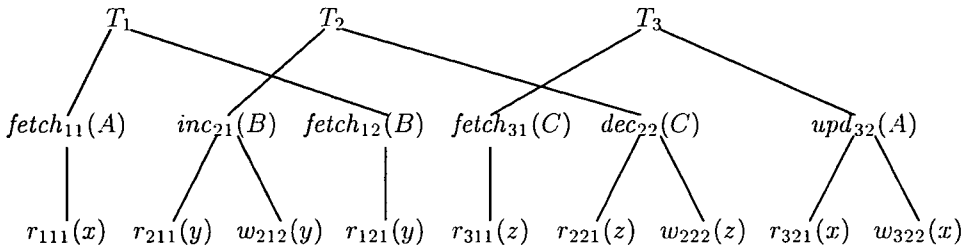

Figure 9: Non-serializable multi-level schedule with str-2PL-overhead

Note that the overhead for FoPL$^+$ occurring in Example 22 is only possible, if we have concurrent access to the same object. On the other hand, the overhead is not as large as expected. Again, the decisive parameter is the number of different operations accessing the same object x within a short period of time.

Example 23 Now consider the schedule in Figure 9. Here str-2PL would request the following locks on L_1-objects:

A	$fetch_1$ upd_3
B	inc_2 $fetch_1$
C	$fetch_3$ dec_2

Since all these pairs of locks are incompatible, the request for the second lock would be rejected. This leads to a deadlock, which can be resolved by aborting and restarting T_3. In addition, $fetch_{12}(B)$ and $dec_{22}(C)$ could first be started after this abort. This means that four L_1-operations had to be repeated. These were composed from six L_0-operations.

If FoPL$^+$ were taken instead, this would result in the following flaglists:

ZL_A	$fetch_1$ upd_3
ZL_B	inc_2 $fetch_1$
ZL_C	$fetch_3$ dec_2

As in the previous example we have to abort and redo $fetch_{31}(A)$, $upd_{32}(A)$ and $dec_{22}(A)$, i.e. three L_1-operations composed from five L_0-operations.

However, with could apply the absorption optimization to commit $dec_{22}(A)$ and hence T_2 immediately. In addition, T_1 would also commit. Since the flag $fetch_3$ would be removed, $fetch_{31}(A)$, and $upd_{32}(A)$ still must be aborted and redone, but this causes only two L_1-operations composed from three L_0-operations.

Finally, since T_2 validates before $upd_{32}(A)$ started, we could even apply early rollback. This means that only $fetch_{31}(A)$ would be repeated, i.e. one L_1-operation composed from one L_0-operation. With these optimizations the overhead caused by str-2PL occurs to be even worse. □

It is not yet possible to draw a general conclusion from these three examples in the sense that FoPL$^+$ is preferable. We could only see, that FoPL$^+$ had advantages, if no abort occurs or an abort occurs for both FoPL$^+$ and str-2PL. Only the situation,

where FoPL$^+$ acted "too optimistically" lead to slight advantages for str-2PL. In order to base such investigations on solid theoretical grounds, a probabilistic model for multi-level transactions must be used.

7 Recovery

In our discussion of concurrency control protocols in the preceding three sections we always provided the necessicity of aborting operations or transactions. This means that we have to undo all the effects issued by such operations, which is a significant part of the recovery component. We usually talk of the *rollback* of an operation.

One possible solution to this problem is to employ the principle of write-ahead-logging (WAL), i.e. before updating the database rollback data are stored at some safe place, which is usually a log-file. A accepted good solution based on WAL is ARIES (**A**lgorithm for **R**ecovery and **I**solation **E**xploiting **S**emantics) [16]. and we shall adopt ARIES to our purposes here. We start giving a short list of the fundamental features of ARIES:

- Recording is not restricted to normal transaction processing, but also happens during rollback through so-called *compensation log records* (CLRs), which prevent UNDO-operations to be executed more than once.
- The storage overhead – besides the logging data – is kept small. On each page only the number of the log record which marks the last change to that page has to be stored.
- ARIES supports partial rollbacks through *savepoints* and fast crash recovery through *checkpoints*, at which information about buffered pages are stored.
- ARIES uses only short-term-locks – so-called *latches* – to access pages, whereas long-term-locks as required by locking protocols are reserved for records.

In [22] an extension ARIES/NT of ARIES to nested transactions has been presented. This extension is tighly coupled with locking protocols and does not employ inverse operations, which are possible in multi-level transactions. In particular, locks are not released after finishing operations that are not transactions. The alternative MLR discussed in [15] exploits inverse operations, but unfortunately assumes them to exist in any case. If they do not exist, the restrictions of ARIES/NT are kept.

In the following we present the extension ARIES/ML for multi-level transactions [6]. ARIES/ML is rather close to MLR, but is not necessarily coupled with a locking protocol. Furthermore, we explicitly differentiate between operations for which there exists an inverse and those for which there exists none.

The major features of ARIES will be preserved. We describe necessary extensions to the data structures and their usage during normal processing and rollback.

The extension allows a coupling with a locking protocol and FoPL and provides the necessary extensions to FoPL with respect to operation aborts. In this way we are also able to support crash recovery.

The data structures used in ARIES/ML comprise various types of *log records* stored in the log-file, an *operation table* and a *dirty pages table*. Each log record has a *log serial number* (LSN) and a field indicating its type, which is ULR, CLR, CCR, RCR, CR, SP or CP. Concretely, we distinguish *update log records* (ULRs), *compensation log records* (CLRs), *committed child records* (CCRs), *reactivate child records* (RCRs), *commit records* (CRs), *savepoints* (SPs) and *checkpoints* (CPs).

Update log records are created during normal transaction processing. Compensation log records record UNDO-operations corresponding to some operation. Committed child records are created, when an operation on a level L_i ($i \neq n$) has finished. Reactivate child records are created during rollback; they correspond to CCRs. Commit records are created, when a transaction commits.

Savepoints are only used to support partial rollback. Thus, it is sufficient to provide their LSN and their type. Checkpoints are used to fasten crash recovery. They are created regularly. Besides LSN and type they contain the dirty pages table, the operation table and some additional data about the database files. The actual storage of buffered pages is left to the buffer manager. We dispense with an intensive discussion of savepoints and checkpoints.

7.1 Log Records for Normal Processing

In order to define the structure of these records for an L_i-operation o we assume a total order $\sqsubseteq_i^{(j)}$ on $\mathfrak{O}_i^{(j)}$ that includes $<_i^{(j)}$. For simplicity assume that the indices are chosen in such a way that $o_{j\beta k} \sqsubseteq_i^{(j)} o_{j\beta\ell} \Rightarrow k \leq \ell$ holds.

Definition 15 Let $o = op_{j\alpha k}(x)$ be an L_i-operation ($i \neq n$) of the n-level-transaction T_j. The *update log record* $ulr_{j\alpha k}$ corresponding to o has the form

$$ulr_{j\alpha k} \quad = \quad (lsn_{j\alpha k},\ \text{ULR},\ j\alpha,\ lsn_{j\alpha k-1},\ p,\ eff_{j\alpha k})$$

with the log serial number $lsn_{j\alpha k}$, the type ULR, the identifier $j\alpha$ of the parent operation $trans(o)$, the log serial number $lsn_{j\alpha k-1}$ of the previous operation in $act(o_{j\alpha})$, a pointer p to the page containing the object x affected by o and the effect of o according to Definition 6. □

In general, to refer to the components of a ULR, we write (LSN, type, OpId, PrevLSN, PageId, data). If o is the first operation in $act(o')$, then PrevLSN is undefined, indicated by the null value $\perp$. PageId may also be left undefined, if the object is only virtual, i.e. realized by a set of other objects. Note that ULRs were already present in the basic ARIES algorithms.

CCRs are created, when an L_i-operation o ($0 < i < n$) has finished. Same as ULRs they contain LSN, type, OpId and PrevLSN. Furthermore, they have a field LastLSN containing a pointer to the last log record created by some operation in $act(o)$, a field ChildId containing the identifier of o itself and a field Op containing

the operator of o to indicate, whether a compensation will be possible or not. Thus, we may write (LSN, type, OpId, LastLSN, ChildId, LastLSN, Op).

Definition 16 Let $o = op_{j\alpha k}(x)$ be an L_i-operation $(0 < i < n)$ of the n-level-transaction T_j. The *committed child record* $ccr_{j\alpha k}$ corresponding to o has the form

$$ccr_{j\alpha k} \quad = \quad (lsn_{j\alpha},\ \text{CCR},\ j\alpha,\ lsn_{j\alpha k\ell},\ j\alpha k,\ lsn_{j\alpha k-1},\ op)$$

with the log serial number $lsn_{j\alpha}$, the type CCR, the identifier $j\alpha$ of the parent operation $trans(o)$, the log serial number $lsn_{j\alpha k\ell}$ corresponding to the last operation in $act(o_{j\alpha k})$, the identifier $j\alpha k$ of the operation itself, the log serial number $lsn_{j\alpha k-1}$ of the previous operation in $act(o_{j\alpha})$ and the operator op. □

Commit records are created, when a transaction T_j commits. They are described by LSN, type and OpId. Formally, a *commit record* for an n-level transaction T_j has the form $cr_j = (lsn_j,\ \text{CR},\ j,\ lsn_{jk})$ with the meaning of these components as in Definition 15 before.

7.2 Log Records for UNDO

Since CLRs record UNDO-operations, they also contain LSN, type, OpId, PrevLSN, PageId and a field containing the data which is necessary for REDO. This can be either a before image expressed by the effect as in ULRs or a compensation operation. In addition, CLRs have a field UNDOnextLSN containing the LSN of the log record for the next operation to be undone. Thus, we have the form (LSN, type, OpId, PrevLSN, UNDOnextLSN, PageId, data)

Definition 17 Let $o = o_{j\alpha k}(x)$ be an L_i-operation $(i \neq n)$ of the n-level-transaction T_j. A *compensation log record* $clr_{j\alpha k}$ corresponding to o has the form

$$clr_{j\alpha k} \quad = \quad (lsn^{clr}_{j\alpha k},\ \text{CLR},\ j\alpha k,\ lsn_{j\alpha k},\ lsn^{clr}_{j\alpha k-1},\ p,\ d)$$

with the log serial number $lsn^{clr}_{j\alpha k}$, the type CLR, the identifier $j\alpha k$ of the rolled back operation, the log serial number $lsn_{j\alpha k-1}$ of the ULR for the previous operation in $act(o_{j\alpha})$ the log serial number $lsn^{clr}_{j\alpha k-1}$ of the log record for the next operation to be undone and a pointer p to the page containing the object x affected by o. The last field d is either the effect $eff_{j\alpha k}$ of o according to Definition 6 or a compensation operation o^{-1}. □

CLRs existed already in ARIES. The only difference here is that the data part of a CLR may now contain a compensation operation, unless o resides on level L_0.

Reactivate child records are also created during rollback, when a finished L_i-operation has to be reinstalled in the operation table. Besides LSN and type a RCR has fields OpId, PrevLSN, ChildId, LastLSN and UNDOnextLSN with the same meaning as for the other kinds of log records.

7.3 Normal Transaction Processing

During normal transaction processing the corresponding ULRs, CCRs, CRs, SPs and CPs are written into the log-file. In addition, each page will contain a field PageLSN, in which the LSN of the last entry writing to that page is recorded. For page access, latches are used also by ARIES/ML.

Finally, ARIES/ML manages an operation table and a dirty pages table. The operation table contains information about active operations. Each record in this table contains

- an operation identifier OpId,
- the status of that operation, which may be 'propagate' (p), 'validate' (v) – not used with locking protocols – 'commit' (c) or 'abort' (a),
- LastLSN and UNDOnextLSN.

Whenever a CCR is created the corresponding operation does not need to be kept in the operation table. The same applies to CRs for top-level operations, i.e. transactions.

The dirty pages table contains information about buffered pages. Each of its records contains a PageId and a recovery LSN (RecLSN), which marks the first entry in the log file from which updates to that page were not yet made persistent.

Example 24 Consider the schedule from Figure 1. Assume that x is stored on page p, y on page q and that p is made persistent by the buffer manager after finishing o_{21}. Then the log records in the following list will be created. The list also indicates the dirty pages table (abbreviated as d.p.t.), the operation table and the pair of PageLSNs for p and q.

log-entry	operation table	d.p.t.	Page LSNs
(1,ULR,11,⊥,p, ...)	(1,p,⊥,⊥) (11,p,1,1)		(⊥,⊥)
(2,ULR,11,1,p, ...)	(1,p,⊥,⊥) (11,p,2,2)	(p,2)	(2,⊥)
(3,CCR,1,⊥,11,2,*inc*)	(1,p,3,3)	(p,2)	(2,⊥)
(4,ULR,21,⊥,p, ...)	(1,p,3,3) (2,p,⊥,⊥) (21,p,4,4)	(p,2)	(2,⊥)
(5,ULR,21,4,p, ...)	(1,p,3,3) (2,p,⊥,⊥) (21,p,5,5)	(p,2)	(2,⊥)
(6,CCR,2,⊥,21,5,*inc*)	(1,p,3,3) (2,p,6,6)		(⊥,⊥)
(7,ULR,22,⊥,q, ...)	(1,p,3,3) (2,p,6,6) (22,p,7,7)	(q,7)	(⊥,7)
...	...	...	...

Dots indicate some data which are left unspecified. □

7.4 Rollback

Rollback may be started at any time and can be executed until a specified savepoint is reached. Thus, to start a rollback we need a set OpIdSet of operation identifiers and a SaveLSN with SaveLSN = 0 corresponding to a complete rollback.

The first activity is to create a *rollback list* containing the LastLSN from all active operations with a parent in OpIdSet. For this the operation table has to be accessed. Then UNDO-operations will be processed by decreasing LSN following the PrevLSN-entries in log records. Only LSNs that are larger than the given SaveLSN will be considered. Rollback stops, when the rollback list becomes empty.

Depending on the type and the content of the log record r with LSN in the rollback list different actions will be triggered:

- In the case $r.type$ = ULR an UNDO-operation will be performed and the PageLSN of the page affected by the operation underlying r will be reset. The necessary data are kept in the ULR. Furthermore, r.PrevLSN will be added to the rollback list and a CLR r' with r'.UNDOnextLSN = r.PrevLSN will be created. Finally, the fields LastLSN and UNDOnextLSN in the corresponding operation table record will be updated. In this case there is no difference to ARIES.

- In the case $r.type$ = CCR we have to distinguish two different subcases.

  If there exists a compensation operation, it will be executed. If we assume a locking protocol for concurrency control, there is a risk for deadlocks now. ARIES/ML circumvents this problem by allowing only one compensation operation to be active. If it is involved in a deadlock, one of the other operations will be chosen for abort. Thus, in this subcase there is not a big difference to the ULR-case before. In particular, a single CLR will be created.

  Now assume that there is no compensation operation. In this subcase the child operation has to be reactivated and an RCR will be created. Both LastLSN and PrevLSN give rise to new entries in the rollback list.

- The cases $r.type$ = CLR and $r.type$ = RCR can only occur, if a partial rollback has already been performed. In both cases there is nothing to do; just add PrevLSN to the rollback list.

Example 25 Consider the following sequence of log records:

(1,ULR,111,⊥, ...) (2,CCR,11,⊥,111,1, ...) (3,ULR,112,⊥, ...)
(4,CCR,11,2,112,3, ...) (5,CCR,1,⊥,11,4, ...) (6,ULR,121,⊥, ...)
(7,ULR,121,6, ...) (8,CCR,12,⊥,121,7, ...) (9,ULR,122,⊥, ...)
(10,CCR,12,8,122,9, ...) (11,ULR,123,⊥, ...) ,

where the underlined type CCR refers to a compensable operation and dots are used to indicate page identifiers and data entries we are not interested in in this example. A complete rollback of o_1 will start with the rollback list (11,10,5) and create the following continuation of the log sequence:

(12,CLR,123,11,⊥, ...) (13,CLR,12,10,8, ...) (14,RCR,12,13,121,7,⊥)
(15,CLR,121,7,6, ...) (16,CLR,121,15,⊥, ...) (17,CLR,1,5,⊥, ...) .

Here the fifth field in CLRs contains the UNDOnextLSN. Fields in RCRs are listed in the order described above. □

7.5 Crash Recovery

Crash recovery in ARIES/ML follows the same ground procedure as ARIES, i.e. we have three consecutive passes for analysis, REDO and UNDO.

The analysis pass is based on log records starting from the last checkpoint. The goal is to discover where to start the REDO-pass and the set of operations to be undone. The last checkpoint allows an initial reconstruction of the operation table and the dirty pages table. Then log records r following the checkpoint entry are read one after the other. Depending on the type and content of r different actions will be triggered:

- If r.OpId exists, then an entry for OpId must be added to the operation table unless a corresponding record exists. In both cases, the LastLSN will be set to r.LSN.
- If $r.type$ = ULR or $r.type$ = CLR, then the dirty pages table may contain a wrong RecLSN entry for the page indicated by r.PageId. If this is the case RecLSN will be set to r.LSN.
- If $r.type$ = CCR, then the entry for r.ChildId will be deleted in the operation table.
- If $r.type$ = RCR, then (r.ChildId, p, r.LastLSN, r.LastLSN) has to be added to the operation table.
- If $r.type$ = CR, then the entry for r.OpId has to be removed from the operation table.

After analysing these log records, the starting LSN for the REDO-pass will be set to the minimum of all RecLSNs in the dirty pages table. The set OpIdSet of operations to be undone contains all operation identifiers from the operation table which do not have the status 'commit'.

For the REDO-pass there are no changes to ARIES, i.e. log records r starting from REDO-LSN as discovered in the analysis pass will be excuted again, if r.PageId occurs in the dirty pages table and RecLSN $\leq r$.LSN $\wedge$ PageLSN $< r$.LSN holds.

In the UNDO-pass ARIES/ML starts a complete rollback with OpIdSet from the analysis pass and SaveLSN = 0.

8 Conclusion

In this paper we investigated concurrency control and recovery for multi-level transactions which occur naturally in distributed databases. The general idea is to exploit application semantics to reduce the number of conflicts.

Two-phase locking (2PL) can be easily generalized to the multi-level case keeping the advantages of locking protocols. All schedules accepted by 2PL will be serializable. Furthermore, strict 2PL leads to schedules that are recoverable and

strict on all levels. As with locking for flat transaction systems the major drawback results from the possibility of deadlocks with the well-known time-consuming detection algorithms.

As an alternative we developed the hybrid FoPL protocol (Forward oriented Concurrency Control with Preordered Locking). Same as 2PL, FoPL only accepts serializable schedules. If combined with a waiting strategy for the case of not successful validation (lazy abort), the modified FoPL$^+$ protocol will accept all serializable schedules. Possible deadlocks in the waiting graph are not critical, since objects are not locked. Moreover, the accepted schedules will be recoverable on all levels. In contrast to 2PL the FoPL protocol is deadlock-free. However, as with other optimistic or hybrid protocols strictness nor cascade-freeness cannot be guaranteed. Finally, we were able to discuss several optimizations of the basic FoPL protocol.

Which choice – strict 2PL or FoPL/FoPL$^+$ – is the better one, depends on various factors. The most important one concerns the probability of conflicts. In general, it is assumed – and this is one of the major motivations behind multi-level transactions – that at least on higher levels the conflict rate will tremendously decrease, which is an argument favouring FoPL. We currently start to realize a test bed in order to compare transaction throughput for various multi-level protocols. We plan to extend these examinations also to generalizations of hybrid protocols that employ time-stamps [3, 10].

The basic idea underlying FoPL stems from the ODL (Optimistic Dummy Lock) protocol [11]. Therefore, it is worth to spend a few words on a comparison. Since ODL has been developed for flat transactions, we must base this comparison on this special case. ODL also uses flags – the so-called "dummy locks" – in the propagation phase. When a transaction T_j issues a read-operation on object x, a *flag* F_j is set on the object x. F_j can be deleted by T_j itself during its validation phase or by another transaction T_k, when T_k performs an actual write-operation on x. Validation basically consists in checking, whether flags are still set.

Compared with FoPL (applied to 1-level-transactions) the major differences are that FoPL uses flaglists, whereas ODL uses a single flag, and that ODL employs a backward validation strategy. Thus, for each commit ODL will force all other operations accessing the same object to abort, no matter whether this is necessary or not. Furthermore, as shown in [21] the backward validation strategy makes a generalization of ODL to multi-level transactions nearly impossible.

As to recovery we adapted ARIES [16] to work both with multi-level locking protocols and FoPL. In the former case one crucial point was to avoid deadlocks during rollback. The extension ARIES/ML preserves the advantages of ARIES such as partial rollbacks, different locking granularities, small storage overhead and the avoidance of multiple UNDO.

References

[1] C. Beeri, A. Bernstein, N. Goodman. A Model for Concurrency in Nested

Transactions Systems. *Journal of the ACM*, 36(2) : 230–269, 1989.

[2] B. Bhargava. Performance Evaluation of the Optimistic Approach to Distributed Database Systems and its Comparison to Locking. In *Proc. 3rd Intern. Conference on Distributed Computing Systems*. IEEE 1982.

[3] C. Boksenbaum, M. Cart, J. Ferrie, J.-F. Pons. Concurrent Certifications by Intervals of Timestamps in Distributed Database Systems. *IEEE Transactions on Software Engineering*, 13(4) : 409–419, 1987.

[4] A. Bernstein, N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Transactions on Computer Systems*, 13(2) : 121–157, 1981.

[5] K. Chandry, J. Misra. A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems. In *Proc. ACM Conference on Principles of Distributed Computing* : 157–164, 1982.

[6] S. Drechsler. *Kopplung des ARIES-Recovery-Systems mit hybriden Mehrschichtschedulern*. Master Thesis, Clausthal Technical University, 1998.

[7] A. Elmargamid, Y. Leu. An Optimistic Concurrency Control Algorithm for Heterogenous Distributed Database Systems. *IEEE Transactions on Data and Knowledge Engineering*, 10(6) : 26–32, 1987.

[8] J. Gray, A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishing, 1993.

[9] T. Härder. Observations on Optimistic Concurrency Control Schemes. *Information Systems*, 9(2) : 111–120, 1984.

[10] U. Halici, A. Dogac. Concurrency Control in Distributed Databases Through Time Intervals and Short-Term Locks. *IEEE Transactions on Software Engineering*, 12(8) : 994–1003, 1989.

[11] U. Halici, A. Dogac. An Optimistic Locking Technique for Concurrency Control in Distributed Databases. *IEEE Transactions on Software Engineering*, 17(7) : 712–124, 1991.

[12] H. Kung, J. Robinson. On Optimistic Methods for Concurrency Control. In *Proceedings of the 5th VLDB-Conference*, 1979.

[13] V. Li. Performance Model of Timestamp Ordering Concurrency Control Algorithms in Distributed Databases. *IEEE Transactions on Computing*, 1987.

[14] W.-T. Lin, J. Nolte. Basic Timestamping, Multiple Version Timestamp, and Two-Phase Locking. In *Proccedings of the 9th VLDB-Conference* : 109–119, 1983.

[15] D. B. Lomet. MLR: A Recovery Method for Multi-level Systems. In M. Stonebraker (Ed.). *Proc. SIGMOD 1992*: 185–194, San Diego 1992.

[16] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write Ahead Logging. *ACM Transactions on Database Systems*, 17(1) : 94–162, 1992.

[17] C. Mohan, B. Lindsay, R. Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems*, 11(4), 1986.

[18] R. Obermarck. Distributed Deadlock Detection Algorithm. *ACM Transactions on Database Systems*, 7(2), 1982.

[19] M. Özsu, P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall International, 1994.

[20] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[21] T. Ripke. *Verteilte hybride Synchronisationstechniken in Mehrschichten-Transaktionssystemen*. Ph.D. Thesis. Clausthal Technical University, 1998.

[22] K. Rothermel, C. Mohan. ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions. In P. M. G. Apers, G. Wiederhold (Eds.). *Proc. 15th VLDB*: 337–346, Amsterdam 1989.

[23] G. Weikum. *Transaktionsverwaltung in Datenbanksystemen mit Schichtenarchitektur*. Ph.D. Thesis. Darmstadt Technical University, 1986.

[24] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems*, 16(1):132–180, 1991.

Received February, 1999