Chapter 7

Concurrency Control Algorithms on Objects

"A journey of thousand miles must begin with a single step." (Lao-tse)

7.1 Goal and Overview of the Chapter

In this chapter we will introduce practical concurrency control algorithms for object-model-style systems, with particular focus again on the important special case of layered systems. We will show that the presented protocols are correct in that they produce tree-reducible schedules only, or even the more restricted class of schedules where all level-to-level schedules are OCSR or CSR and conflict-faithful. As the latter kind of decomposition already suggests, we will see that the conventional page-model protocols can be used as building blocks in the construction of object-model protocols. For the most part, we will even concentrate on locking protocols and particularly 2PL.

Throughout this chapter we will assume that commutativity is specified as a simple table on a per-object type or on a per-layer basis. Thus, operations on different object types can safely be assumed to be commutative, unless — as we will see — one object type serves to implement the other, which will cause some complications. We will mostly focus on exploiting state-independent commutativity, and consider return-value commutativity only in the final section of this chapter.

7.2 Locking for Flat Object Transactions

The commutativity table for an object can directly be interpreted as a lock mode compatibility table. For each operation $f(x_1, \ldots, x_n)$ with input parameters x_1, $\ldots$, x_n a lock is acquired in lock mode f on items $x_{i_1}, \ldots, x_{i_m}$ $(m \leq n)$, where the $x_{i_1}, \ldots, x_{i_m}$ form a subset of the parameters relevant for deciding on conflicts. Most often, m will be 1. This case can be interpreted as follows: there is a "primary" object to which the operation refers; operations on different primary objects imply commutativity, whereas commutativity on the same object depends only on the operation type, not on the other input parameters.

As for the actual locking protocol, i.e., the discipline according to which locks are acquired and released, we can apply 2PL, S2PL or SS2PL to the enhanced setting of object-model operations in a straightfoward way, and all correctness arguments also carry over in an almost trivial way. Note, however, that the object-level operations themselves need to be executed strictly sequentially (i.e., concurrency is limited to interleavings at the highest level below the transaction roots). The reason is that flat object schedules always assume that their object-level operations are indivisible (see Chapter 6).

Figure 7.1 shows an example of two concurrent transactions, consisting of deposit and withdraw operations, under the S2PL protocol driven by a commutativity-based lock-mode conflict table. As in previous illustrations of this style, dashed lines denote lock waits. Transaction t_2, which requests a withdraw lock on account a has to wait until t_1, which holds an incompatible deposit lock on a, commits and releases its locks.

7.3 Layered Locking

When higher-level operations can be executed concurrently on lower levels, one way to ensure tree-reducibility is to enforce the order-preserving conflict-serializability (OCSR) of the level-to-level schedules. This can in turn be guaranteed by applying an appropriate "flat" concurrrency control protocol between each pair of adjacent levels. One suitable protocol is 2PL or its derivatives S2PL and SS2PL; in fact, this family provides the most versatile and robust protocols that are known for flat schedules and are therefore our methods of choice.

inner-node operations as subtransactions

When 2PL is applied to layered schedules, we arrive at a nicely modular archi-

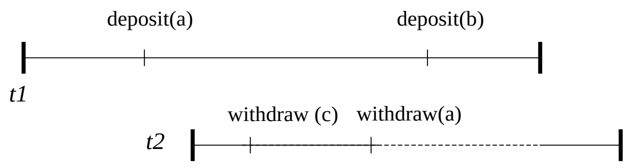

Figure 7.1: 2PL applied to a flat object schedule.

tecture: the scheduler between level L_{i+1} and L_i needs to know only the operations at level L_i and the identifiers of their L_{i+1} parents. From the viewpoint of this scheduler, the L_{i+1} operations are transactions and the L_i operations are the steps of these transactions. To better distinguish this role of L_{i+1} operations in a level-to-level schedule from the actual transaction roots, we will refer to them in the current context as *subtransactions*.

After these considerations, we are ready to give the rules for what we call *layered 2PL* for n-level schedules:

layered 2PL

1. *Lock acquisition rule:*
   When an L_i operation $f(x)$ with parent p is issued, an f-mode lock on x needs to be acquired before the operation can start its execution. This lock is called an L_i lock.

2. *Lock release rule:*
   Once an L_i lock, originally acquired by an operation $f(x)$ with parent p, is released, no other child of p is allowed to acquire any L_i locks.

3. *Subtransaction rule:*
   Upon the termination of an L_i operation $f(x)$, all locks at level L_{i-1} that have been acquired for the children of $f(x)$ are released.

So the key principle of layered 2PL is that locks at a certain level, say L_i, are held on behalf of the corresponding subtransactions, the L_{i+1} operations. With SS2PL, this amounts to holding locks for the duration of a subtransaction, but no longer. It is exactly this "early release" of low-level locks such as page locks that makes layered 2PL such a powerful protocol. In terms of the possible concurrency, it is vastly

superior to the conventional page-model 2PL and other page-model concurrency control methods..

As for the correctness of layered 2PL, we can now leverage the theory developed in Chapter 6 and obtain the following correctness proof with little effort:

Theorem 7.1

Layered 2PL generates tree-reducible schedules only.

Proof. All level-to-level schedules are OCSR, hence the schedule is tree-reducible according to Theorem 6.15. □

To illustrate the layered 2PL protocol, reconsider the example from Figure 6.4 with operations on the record and page levels. Figure 7.2 shows a possible execution of this schedule under layered 2PL. This is an example for employing record-level locking in combination with short-term page locking for the duration of subtransactions, an extremely important case in practice. The subtransactions that define the scope of the page locks are highlighted by the grey triangles. As usual, lock waits are indicated by dashed lines, and we see that such a protocol may incur both record-level and page-level lock waits. Moreover, we also see that page lock waits are much shorter than they would be under pure page-level locking, where page locks would be held for the duration of the entire transactions.

special case: page latches for single-page record operations

A special case of the above setting would be a system where all record-level operations access only a single page, like the $fetch(x)$ operation in Figure 7.2. If we knew that all operations on a given set of records have such a structure, the necessary subtransaction-duration 2PL can be further streamlined. It would be sufficient to "latch" the affected page during the record operation, by atomically setting or resetting a bit in the page header, rather than acquiring a "full-fledged" page lock. The functionality of latching is similar to that of semaphores provided by an operating system, but latching avoids expensive operating-system calls as much as possible. In comparison to explicit locking, latching does not provide higher concurrency, but its overhead is much lower than that of a lock manager. For example, latching does not keep a queue of waiting requestors, nor does it need to check for deadlocks. For this reason, commercial database system use such latching techniques whenever they are applicable. However, as the example shows, storage structures can be fairly complex and may require multi-page record-level operations, especially when deal-

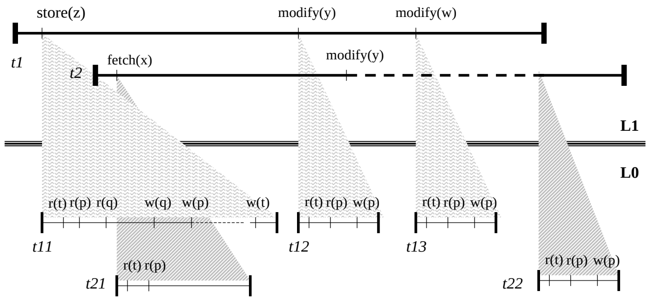

Figure 7.2: Layered 2PL applied to the record and page levels of a 2-level schedule.

ing with hierarchically structured records as storage representations of composite objects or XML documents in object-relational database systems.

opportunities for index locking

So latching surely has its limits. In particular, latching is not easily applicable to index operations such as searching a key in a B^+-tree, which involves multiple pages in the tree traversal. Obviously, the situation becomes even more complicated for insertions and deletions of (key, RID) pairs, as this may trigger node splits in the B^+-tree. Nevertheless, this case is still special in that it leads to a limited class of page-access patterns only, so that specialized concurrency control protocols could be attractive for such subtransactions (e.g., the tree locking protocol from Chapter 4). As index management is among the most important issues in data management with regard to performance, we postpone the details of this discussion and will look into this issue in a separate chapter, Chapter 9.

special case: all-commutative chained subtransactions

Another special case would be the situation where all object-level operations are a priori known to commute. Then no object-level locks would be necessary at all, and transactions are effectively decomposed into sequences of so-called *chained subtransactions*. Locking would be strictly confined to the duration of subtransactions so that very high concurrency is possible. The caveat with regard to this approach is that one needs to design the application very carefully to ensure an all-commutative object layer. Typically, this is feasible only for very specialized, functionally restricted applications. An example would be the inter-bank funds-transfer application known as SWIFT, where all withdraw and deposit operations are declared to be

commutative or, actually, "compatible", tolerating the application-wise "negligible" differences in the effect of differently ordered withdrawals. Note, of course, that this system would typically be used only for relatively small amounts of money, and also relies on the legal rules and mutual guarantees for inter-bank business.

Layered 2PL is, of course, applicable to more than two levels as well. As an example, consider the 3-level schedule shown in Figure 7.3. It shows SQL-style query-level operations at level L_2, which are translated into record and index operations at level L_1, which finally result in page-level read and write operations. Recall from Chapter 1 that L_2 insert operations need to maintain all indexes at level L_1, namely indexes on the Age and City attributes of the Persons table in the example. Queries, on the other hand, may exploit multiple indexes by looking up RID lists for the specified values of different attributes, intersecting the RID lists, and finally fetching only the records that satisfy the conjunction of the specified conditions.

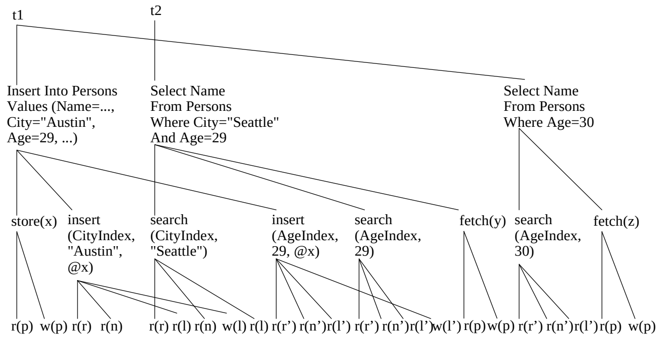

Figure 7.3: Example of a 3-level schedule with query, record, and page level.

The execution that results from this schedule under a 3-level layered 2PL is shown in Figure 7.4. Subtransactions are again highlighted as shaded triangles, with different shading patterns used for the two transactions of the example. Here we assume that the locks at the query level L_2 are some form of so-called *predicate locks*. The lock for the select operation of t_2 essentially locks the conjunctive condition *City="Seattle" and Age = 29*, whereas t_1's select operation locks the condition *Age = 30* and t_1's insert operation essentially locks the conjunction of the inserted record's attribute values, namely, *Name = ... and City = "Austin" and Age = 29 and ...*. A lock conflict exists whenever the predicates of two lock requests intersect. More precisely, this means that the conjunction of the two predicates is satisfiable in that it could possibly be satisfied by some record regardless of whether this record does actually exist in the database or not. Conversely, when the predicates of two lock requests are disjoint in that their conjunction evaluates to false and is thus insatisfiable, we can safely assume that no conflict exists at the query level L_2. We will further elaborate on the concept of predicate locking in Chapter 8. In the example, no predicate-lock conflict exists at L_2, whereas conflicts arise for the index search keys at L_1 and incur lock waits. Note the lock wait of t_{21} in Figure 7.4 when it requests a lock for the operation $search(AgeIndex, 29)$ at level $L1$. t_{21} has to wait until the conflicting lock for operation $insert(AgeIndex, 29, @x)$ is released at the end of t_{11}. For this reason, t_{212} is forced to sequentially follow t_{113}, which explains the difference in the interleavings between these two subtransactions compared to the (input) schedule shown in Figure 7.3.

Layered 2PL can hence be applied to an arbitrary number of levels. However, although this may, in principle, be beneficial for concurrency, the inevitable overhead for the lock management suggests to keep the number of levels rather small, typically two or three. When given an n-level schedule with large n, the question is whether we can apply layered 2PL to a limited subset of the levels and still guarantee that the schedule will be tree-reducible. It turns out that locks do not need to be acquired at all layers of such an n-level schedule. Rather, one can skip certain layers, say layer L_i, by extending the scope of the subtransactions above L_i, which effectively leads to longer lock durations of the locks below L_i. So, for example, L_{i-1} locks could be held for the duration of the corresponding operations' grandparents, i.e., the L_{i+1} subtransactions. In essence, this removes an entire layer from the schedule itself, as far as concurrency control is concerned, and extends the caller-callee relationship by connecting the L_{i-1} nodes in the transaction tree directly to their grandparents. Once this is accomplished, we have again a layered schedule with one level less, and we can apply our standard reasoning for layered schedules.

This consideration leads to a *selective layered 2PL* with the following rules, where

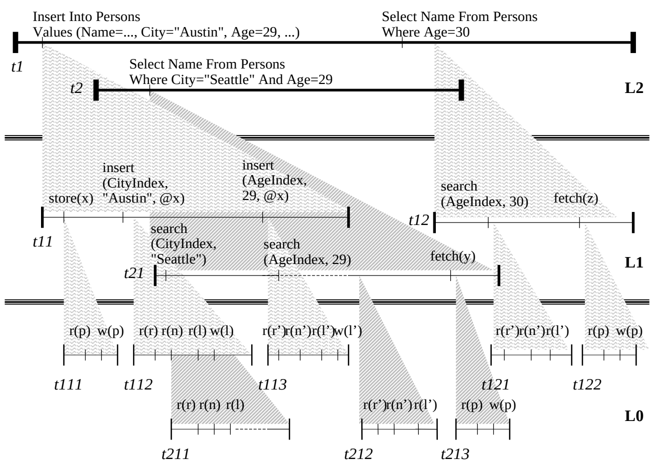

Figure 7.4: Layered 2PL applied to the query, record, and page levels of the 3-level schedule of Figure 7.3.

out of the $n+1$ layers $L_n, \ldots, L_0$ of an n-level schedule a subset $L_{i_0}, \ldots L_{i_k}$, with $1 \leq k \leq n$, $i_\nu > i_{\nu+1}$, and $i_0 = n, i_k = 0$, can be selected with the following locking rules (which are merely slight modifications of the original layered 2PL rules):

1. *Lock acquisition rule:*
   When an L_{i_ν} $(\nu > 0)$ operation $f(x)$ with $L_{i_{\nu-1}}$ ancestor p is issued, an f-mode lock on x needs to be acquired before the operation can start its execution. This lock is called an L_{i_ν} lock.

2. *Lock release rule:*
   Once an L_{i_ν} lock, originally acquired by an operation $f(x)$ with $L_{i_{\nu-1}}$ ancestor p, is released, no other L_{i_ν} descendant of p is allowed to acquire any L_i locks.

3. *Subtransaction rule:*
   At the termination of an L_{i_ν} operation $f(x)$, all locks at level $L_{i_{\nu+1}}$ that have been acquired for the $L_{i_{\nu+1}}$ descendants of $f(x)$ are released.

Reconsider the example of Figure 7.3, whose resulting 3-level layered 2PL execution was shown in Figure 7.4. With selective layered 2PL, applied to the query level $L2$ and the page level L_0, the resulting execution keeps page-level locks longer and leads to a longer lock wait for the L_0 operation $r_{211}(l)$, as shown in Figure 7.5.

layered 2PL with intra-transaction parallelism

We have seen that a given layer manages the locks on behalf of the subtransactions that is sees, regardless of the further ancestors of these subtransactions. So a page-level lock manager that handles subtransactions each of which corresponds to a record-level operation does not even need to know to which transactions the various subtransactions belong. Thus, a nice observation is that the same lock manager could handle parallel subtransactions that originate from the same transaction without any additional means. This form of *intra-transaction parallelism* is, of course, only possible if the higher-level operations are conflict-free and the transaction program or the server's query-processing engine does not require any execution precedence among such high-level operations. When subtransactions are issued in parallel, the underlying layer will guarantee equivalence to some serial order, and this works even for parallelized update statements. As an example, consider the scenario shown in Figure 7.6.
In this scenario, transaction t_1 issues two subtransactions, t_{14} and t_{15} in parallel, to reflect the modification of record x in the index on the City attribute. The RID for x is removed from the RID list associated with key "Austin" and added to the RID list for "Dallas". These two operations do not conflict conceptually, but they do have a page-level conflict on page l (i.e., the two keys happen to reside on the same leaf page of the index). The protocol takes care of this conflict by blocking one of the two subtransactions for a short while. In addition, a third subtransaction,

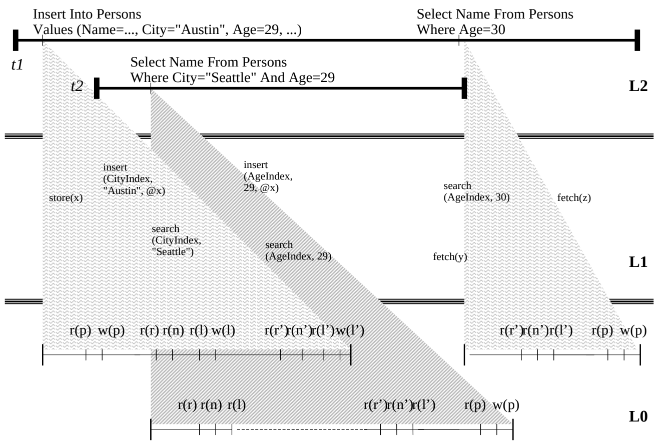

Figure 7.5: Selective layered 2PL applied to the query and page levels of the schedule of Figure 7.3.

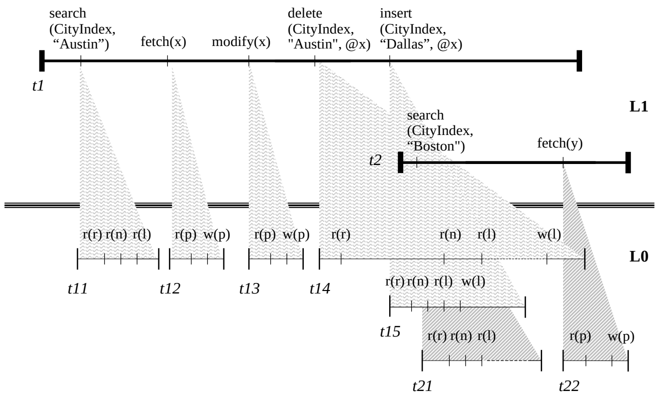

Figure 7.6: Layered 2PL with intra-transaction parallelism.

t_{21}, issued by another transaction, t_2, has a similar conflict and exhibits a short lock wait, too. Note that without page locks (or some equivalent, carefully designed, latching protocol) intra-transaction parallelism for update statements would not be feasible despite the fact that there is no conflict at level $L1$, as t_{14} and t_{15} could then interleave in an arbitrary way and render the index inconsistent.

7.4 Locking on General Transaction Forests

When trying to generalize the layered 2PL to non-layered schedules, the main problem is exactly the missing layering and the resulting difficulty in identifying the appropriate scopes of locking. As an example, consider the schedule in Figure 7.7, where account objects are assumed to be implemented by a counter for the account balance and a list for recording all cash flows on the account. The decisive point is that t_2 "bypasses" the account abstraction and chooses to manipulate the counter and the list directly (see Chapter 6 for a discussion of why and how such bypassing may occur). At the time when the second transaction t_2 starts, we would expect t_1 to hold a deposit lock on account x and no other locks. The increment and append locks on the counter a and the list l would have been released already at the end of

subtransaction t_{11} (i.e., the deposit(x) operation). The page locks for the underlying page reads and writes would have been released even much earlier, at the end of t_{111} and t_{112}, respectively. Now, when t_2 requests an decrement lock on counter a, it would not find any existing locks on a, and the lock request could be granted. The same situation holds for the subsequently needed append lock on list l.

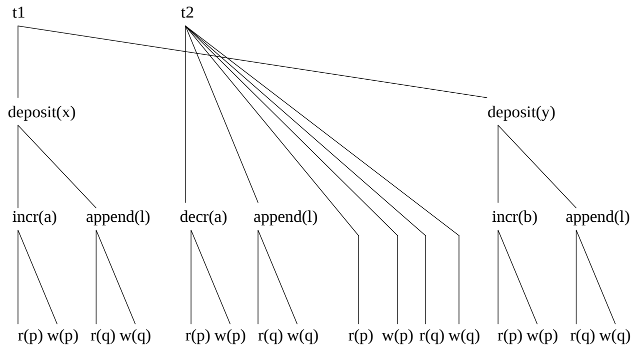

Figure 7.7: Example of a schedule for which layered locking does not work.

The net result is that the schedule shown would be allowed under such a locking protocol. However, this concurrent execution is admissible only under specific assumptions on the semantics of t_2's operations. If we knew that the decrement and append operations of t_2 together implement an abstract operation that commutes with t_1's deposit, then the situation is acceptable. For example, if t_2 decrements the counter by a constant amount to charge a monthly fee for maintaining the account, this would be considered commutative with a deposit. On the other hand, if the decrement and the append together implement a withdrawal, the non-commutativity of withdraw and deposit operations should actually tell the lock manager that t_2 must not be allowed to run at this point. Analogous arguments could be made about the subsequent read/write children of t_2. Without any clue on the abstract semantics of t_2's operation sequence, we would conservatively have to assume a conflict with t_1's high-level deposit operation.

The question now is when we can allow a low-level operation to proceed in a situation where incomparable high-level operations of concurrent transactions still hold locks. The answer lies in inspecting the ancestors of both the requesting transaction and the potentially conflicting high-level operations of the other transactions. If operation r of the requestor (e.g., t_2's decrement) has an ancestor for which we know that it commutes with a previous operation h (e.g., t_1's deposit) and such a condition holds for every transaction that holds an incompatible lock, then operation r can be allowed to proceed; otherwise it should be blocked. Since lock conflicts are actually detected only among operations on the same objects, the negative one of the two cases requires additional information for the previous operations on the object to which the requestor r refers. Essentially we need to know that a lock on that object was held on behalf of a higher-level operation h but has meanwhile been released. For example, t_2's decrement operation should be alerted by remembering the previous lock of t_1's increment on a. Realizing that this is a potentially problematic situation should then lead to a more elaborate conflict test involving the ancestors of the operations.

A technical means for accomplishing the above ancestor-conscious conflict testing is to retain the original locks of completed subtransactions for the purpose of alerting subsequent operations that access a lower-level object directly, bypassing the higher-level abstraction. In an implementation, regular locks would be converted into a special kind of *retained locks* upon the completion of the corresponding subtransaction. Retained locks merely serve the alerting bypassing operations; operations that are invoked on behalf of higher-level operations that have properly passed all conflict tests in a complete top-down descent along an object-implementation hierarchy should never be blocked by a retained lock.

Putting all these considerations together, we arrive at the following principle: An operation r, the requestor, can ignore a retained lock, originally held by operation h, if and only if both operations have ancestors r' and h' such that h' is already completed and r' and h' commute. Otherwise, the retained lock of h is still considered "active" as far as r is concerned. In the worst case, the ancestor of h by which r could become blocked would be the transaction root of h. This would be the case in our example of Figure 7.7: $decr_{21}(a)$ would conflict with the retained lock, originally held by $incr_{11}(a)$, but the only "commutative" ancestors of the two operations are the transactions themselves. So $decr_{21}(a)$ would have to wait until the completion of t_1. Note that it is important that the ancestor of the retained-lock holder h is completed; otherwise, we could end up with subtrees for commutative high-level operations for which we cannot guarantee isolation of the subtrees themselves.

object-model 2PL

We coin this protocol the (general) *object-model 2PL*. It is completely described by the following rules:

1. *Lock acquisition rule:*
   When an operation $f(x)$ with parent p is issued, an f-mode lock on x needs to be acquired before the operation can start its execution.
2. *Lock conflict rule:*
   A lock requested for operation $r(x)$ (called the requestor) is granted only if
   (a) either no conflicting lock on x is held
   (b) or when for every transaction that holds a conflicting lock on behalf of an operation $h(x)$ (called the holder) the following condition is satisfied: the lock of $h(x)$ is a retained lock and r and h have ancestors r' and h', respectively, such that h' is already terminated and commutes with r'.

   In the second case, if no such ancestors exist, $r(x)$ needs to wait until the completion of the lowest ancestor h' of h such that r has an ancestor r' that commutes with h'. In the "worst case", this lowest ancestor h' would be the transaction root to which h belongs.
3. *Lock release rule:*
   Once a lock, originally acquired by an operation $f(x)$ with parent p, is released, no other child of p is allowed to acquire any locks.
4. *Subtransaction rule:*
   At the termination of an operation $f(x)$, all locks that have been acquired for the children of $f(x)$ are converted into retained locks.
5. *Transaction rule:*
   At the termination of a transaction, all locks of its descendants are released.

The correctness arguments for this protocol are quite a bit more involved than those for the layered variant. By proper case analyses and application of the tree-reducibility transformation rules, we arrive at the following result:

Theorem 7.2

The object-model 2PL generates tree-reducible schedules only.

Proof. First assume that all operations in a transaction tree retain all their locks until the commit of the transaction root. In this case each object-model schedule would follow a flattened SS2PL protocol that gradually acquires locks for all nodes of an entire transaction tree and releases them in one stroke at the very end of the transaction. So in reasoning about the correctness of the actual protocol that allows certain operations to ignore retained locks, we can focus on the operations that are in conflict with a retained lock but have an ancestor that commutes with a completed ancestor of the retained lock's holder. So the canonical situation to be inspected more closely is the following: there is a retained lock holder f_1 (somewhere within a transaction tree) that is already completed, has a completed ancestor h_1, and belongs to active transaction t_1, and there is a lock requestor f_2 that would conflict with f_1, belongs to active transaction t_2, and has at least one active ancestor h_2 that commutes with h_1. This situation is depicted in Figure 7.8. All other cases are uncritical because of the locking nature of the protocol. For example, if t_1 was already completed when the lock for f_2 is requested, f_2 could immediately proceed; in this case it is guaranteed that t_1 and t_2 can be isolated from each other by commutativity and tree-pruning arguments such that t_2 follows t_1. Another uncritical situation would be when no ancestor h_2 of f_2 exists that commutes with h_1; in that case, t_2 would have to wait until the completion of t_1.

For the canonical type of critical situation we now show that the subtrees rooted at h_1 and h_2 can always be isolated with both of the two possible serialization orders among them being acceptable. We do so by induction on the longest path from a leaf to h_1 or h_2, whichever yields the maximum.

For the base case with maximum path length 1, both f_1 and f_2 must be leaves, and h_1 and h_2 are their immediate parents. In this case, the flat 2PL protocol for the tree leaves ensures that h_1 and h_2 can be disentangled (if they are concurrent), with h_2 following h_1.

Now suppose that the claim holds for all paths up to length i, and consider h_1 and h_2 such that the longest path from the leaves to one of these operations has length $i+1$. By the induction hypothesis, we can isolate and then prune all subtrees rooted at the children of h_1 and h_2. Now consider two children g_1 and g_2 of h_1 and h_2. If g_1 and g_2 commute, then we can re-arrange their order to whatever is needed for isolating h_1 and h_2. So assume that there is a conflict between g_1 and g_2. Since under the object-model 2PL all conflicts are lock conflicts, one of the two operations must hold a lock that the other operation requests in a conflicting mode. We need to distinguish two cases with respect to the ordering: g_1 precedes g_2 or vice versa. In the first case, if h_1 is still active at the point when g_2 is invoked, g_2

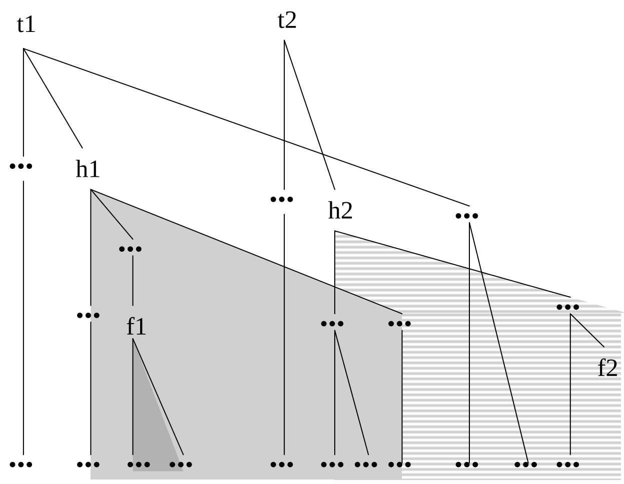

Figure 7.8: Canonical situation for conflict testing against a retained lock under the object-model 2PL protocol.

needs to wait until the completion of h_1; then the 2PL rule would ensure that g_2 can be pushed after the entire subtree of h_1. If h_1 is already terminated at the point when g_2 is invoked, the lock for g_1 must be a retained lock; so g_2 follows the entire subtree of h_1 anyway and does not present any problems in isolating the subtrees of h_1 and h_2. At the next stage, when the subtrees rooted at h_1 and h_2 are pruned, it is perfectly acceptable to forget this serialization ordering between g_1 and g_2, for h_1 and h_2 are known to be commutative. In the second case, where g_2 precedes g_1, the same arguments can be applied analogously. The fact that the g_2 has a descendant f_2 that followed a conflicting descendant f_1 of g_1 does not matter anymore, since, by the induction hypothesis, it was possible to isolate and prune g_1 and g_2 with both possible serialization orders among them being equally acceptable. This completes the induction.

The result about h_1 and h_2 holds for all possible pairs of this kind and all possible path lengths from the leaves. Thus we can now simply specialize h_1 and h_2 to become t_1 and t_2, the transaction roots. So, as a corollary, we obtain that the conflict between f_1 and f_2 does not interfere with the isolation of t_1 and t_2. Note that this does not yet say that t_1 and t_2 can be isolated with either serialization order, as there may be other operations that dictate such an order or may even prevent the isolation. However, those other operations are either of the same kind as f_1 and f_2, or are taken care of by the 2PL rule for non-retained locks. □

A selective form of the protocol, where certain objects in the invocation trees are skipped, is also conceivable, and is analogous to the selective layered 2PL. However, as one does not need to explicitly model each and every invoked object method in the schedule in the first place, the same effect can be achieved by a less comprehensive form of schedule. So the concurrency control would not "see" the less important operations anyway. For this reason and because the selective version of the object-model 2PL becomes notationally complex, we do not further elaborate on it here.

In summary, although the developed general object-model 2PL is a fairly complex protocol, it is worth being employed in situations where the discussed kind of bypassing object abstractions is possible. The alternative would be to resort to standard locking at the common lower-level objects, but this could severely limit the allowable concurrency. As we mentioned already earlier, it is unlikely that the concurrency control is presented with very deep method-invocation trees. So, in a practical application situation, it may also be well possible to simplify and streamline the object-model 2PL to a more restricted setting such as generalized 2-level schedules where the object level L_1 can be bypassed by some transactions.

7.5 Hybrid Algorithms

Different protocols may be used at distinct levels of a layered system. For example, we could use an optimistic protocol or timestamping at the page level and 2PL at the record level. This can be advantageous, as the L_i operation patterns of subtransactions may have more specific characteristics than the patterns of the entire transactions, so that protocols that are specifically geared towards particular patterns could be favored. For example, with record-level and page-level concurrency control in a 2-level system, the fraction of read-only subtransactions at the page level would typically be much higher than the fraction of read-only transactions (i.e., roots).

Thus, particularly attractive hybrid protocols for 2-level schedules are the following:

- a combination of 2PL at the (object or record) level L_1 and the forward-oriented optimistic concurrency control, FOCC, at the page level L_0, or
- a combination of 2PL at L_1 and the multiversion concurrency control protocol that uses timestamping for read-only (sub-)transactions and 2PL for update (sub-)transactions, coined ROMV in Chapter 5, at level L_0.

In both of these combinations, read-only subtransactions are treated particularly favorable and with extremely low overhead. Further note that both FOCC and ROMV can be implemented very easily and efficiently for very short (sub-)transactions on pages, whereas their implementation is much more problematic for longer transactions at the record level.

The correctness of the 2PL/FOCC combination follows more or less directly from the fact that both protocols generate OCSR schedules, thus leading us to the following result:

Theorem 7.3

For 2-level schedules, the combination of 2PL at L_1 and FOCC at L_0 generates tree-reducible schedules only.

Tree-reducibility is also guaranteed by the 2PL/ROMV combination, but as ROMV does not generate CSR L_1-to-L_0 schedules, one needs to use the tree-reducibility transformation rules to show equivalence to a serial order of the roots. The transformation rules can be applied in a strictly layered manner, first isolating record-level operations, then pruning their subtrees, and finally applying CSR-style commutativity arguments at the record layer to isolate transactions. The commutativity-based reordering of the page-level children to isolate the record-level operations is a bit more sophisticated than with 2PL or similar protocols at the page layer. In addition to the usual commutativity relationship, a page-read operation issued by a read-only record-level operation commutes with all page writes whose parents are concurrent with the read operation's parent. The read does not commute, however, with a write step whose parent terminated before the read's parent began. This notion of commutativity reflects the version function of the ROMV protocol. Then it is straightforward to see that record-level operations can be isolated without reversing any orderings among non-concurrent record-level operations, and this yields the following result:

Theorem 7.4

For 2-level schedules, the combination of 2PL at L_1 and ROMV at L_0 generates tree-reducible schedules only.

Obviously, the dual version of this combination with ROMV at the record layer L_1 and 2PL at the page layer L_0 is an intriguing protocol, too. As long as we restrict ourselves to simple fetch and modify operations at the record level, the above arguments are sufficient to prove the correctness of this protocol as well. To incorporate insertions and deletions of records we can assume that deletions actually leave some form of marker in the underlying storage structures rather than removing the record (and its prior versions). With semantically richer operations like predicate-oriented SQL-style search commands, however, we would need to reconsider and extend the notions of a version function and a conflict relation. Thus we have:

Theorem 7.5

For 2-level schedules, the combination of ROMV at L_1, applied to record-oriented $fetch$, $modify$, $store$, and $erase$ operations, and 2PL at L_0 generates tree-reducible schedules only.

In commercial database systems, 2PL at the page layer would often be replaced by a carefully designed latching protocol for reduced overhead.

7.6 Locking for Return-value Commutativity and Escrow Locking

Locking protocols can also be fed by return-value commutativity for defining the compatibility between different lock modes. In this case, each operation type in combination with each of its return value forms a separate lock mode. For example,

for the bank account example, we would introduce lock modes like $withdraw \uparrow ok$, $withdraw \uparrow no$, $deposit \uparrow ok$, and $getbalance \uparrow ok$. The lock mode compatibility would be derived from the table for return-value commutativity in a straightforward way. As before, a lock can only be granted if no other transaction already holds a lock on the same object in an incompatible mode, and, of course, the rules of (layered or object-model) 2PL also need to be in effect.

deferred lock conflict test at end of subtransaction

A problem with this approach is that return values and hence the necessary lock modes are known only at the end of an operation execution; so we cannot know in which mode to acquire the lock for the operation before it starts. The solution is to allow the execution without the appropriate lock; once we know the return value, a lock is requested in the corresponding mode. If there is no lock conflict at this point, the lock is granted and we do not need anything special. On the other hand, if we detect a lock conflict, this means that, in retrospect, we should not have been allowed to execute the operation at all. In this case, the escape is to rollback the operation, request the lock as if the operation were about to start just now, and — because the conflicting lock is already held — put the requesting transaction to wait until the current lock holder releases its lock. Note that it is only the operation that is rolled back, not the entire transaction. The approach can be viewed as an optimistic execution of the subtransaction that corresponds to the operation. Note that it is crucial for this subtransaction to acquire locks for its children to ensure that all of the semantically rich operations for which we want to exploit return-value commutativity are indeed isolated. With this prerequisite, we can show that the sketched protocol guarantees that all resulting schedules are tree-reducible.

A special case of locking based on return-value commutativity has been studied intensively for objects of the abstract data type "counter", a practically important case for reservation systems, inventory control, and other applications along these lines. In these applications counters are typically constrained from both below and/or above. Recall that for counters without any constraints, increment and decrement operations generally commute, and return-value commutativity would not be needed. So each counter object x is associated with a lower bound, $low(x)$, and an upper bound, $high(x)$, within which the value must be guaranteed to stay.

Escrow locking on counter objects

When transactions request to increment or decrement a counter, the locking protocol needs to ensure that the requested operation commutes with all previously executed increment and decrement operations issued by ongoing transactions. As these transactions are still active and as we cannot take for granted that their effects will always be eventually committed, we need to consider a value interval within which the "true" value of a counter is guaranteed to stay, even if some of the active transactions abort. So for each such object x, the concurrency control maintains two values $inf(x)$ and $sup(x)$, the infimum and supremum of the possible value range for

x. Return-value commutativity is then tested against the infimum and supremum of a counter, which can be done very efficiently. This approach is called *Escrow locking*, and the increment and decrement operations are referred to as Escrow operations in this context, as they can be viewed as requesting some quantity of a resource into Escrow, under the supervision of a notary. A concrete application example for this metaphor could be selling or buying stocks for a (constrained) portfolio or maintaining the quantity-on-hand of some item in a warehouse.

Following these considerations, the concurrency-control code for the two Escrow operations *incr* and *decr* is as follows:

```
incr (x, Δ):
  if x.sup + Δ ≤ x.high then
    x.sup := x.sup + Δ; return ok
  else if x.inf + Δ > x.high then
    return no
  else wait fi fi;

decr (x, Δ):
  if x.low ≤ x.inf - Δ then
    x.inf := x.inf - Δ; return ok
  else if x.low > x.sup - Δ then
    return no
  else wait fi fi;
```

In this pseudocode "wait" means waiting until the end of one of the active transactions that has operated on the corresponding counter. Upon the commit or abort of a transaction t the uncertainty interval of a counter value is reduced by increasing the infimum of a counter for each increment operation and decreasing the supremum in the case of a commit, or vice versa when the transaction aborts. To this end, the concurrency control also needs to maintain a list of all increment and decrement operations that have been issued by each active transaction. With this data structure at hand, the code to be executed upon commit or abort looks as follows:


```
commit of transaction t:
  for each operation incr(x,Δ) executed by t do
    x.inf := x.inf + Δ od;
  for each operation decr(x,Δ) executed by t do
    x.sup := x.sup - Δ od;

abort of transaction t:
  for each operation incr(x,Δ) executed by t do
    x.sup := x.sup - Δ od;
  for each operation decr(x,Δ) executed by t do
    x.inf := x.inf + Δ od;
```

When no transaction is active, `x.inf` and `x.sup` for a counter `x` coincide again, yielding the actual value of the counter. An example scenario under the Escrow locking protocol is shown in Figure 7.9, assuming that `x.low` is zero and `x.high` unbounded. In this figure, the vertical dotted lines show the actual value of x at the begin and end of the schedule as well as the infimum and supremum of x at the intermediate points. t_4 is forced to wait until t_2 terminates (by abort), for at the time when t_4 issues its $decr(x, 20)$ operation request `x.inf` is 15 (and `x.sup` equals 75). With the abort of t_2, `x.inf` becomes 25 so that t_4's decrement can safely proceed.

Like most locking protocols, Escrow locking is susceptible to deadlocks. Such a situation is shown in Figure 7.10 where transaction t_4 waits for an Escrow lock but already holds "conventional" locks, and thus causes t_1 and t_2 to become blocked when they request conflicting locks. The resulting lock waits are illustrated by dashed lines. A particularity that arises here is that the lock waits are not simply blocking situations between pairs of transactions. Rather transaction t_4 actually waits for a quantity of 20 of the resource managed by counter x. Once we knew that the infimum of x were at least 20, t_4 could be resumed. With the three still active transactions t_1, t_2, t_3, the infimum is still 0, but it would be sufficient that any two out of these three transactions commit to increase the infimum to 20. So, in essence, t_4 is waiting for any two of the three transactions t_1, t_2, t_3, and t_1 and t_2 each wait for t_4. This form of deadlock would require some kind of "or-edges" in a waiting graph and is more general than the kind of deadlocks that we are used to with conventional locking. In fact, it turns out that detecting that a deadlock exists

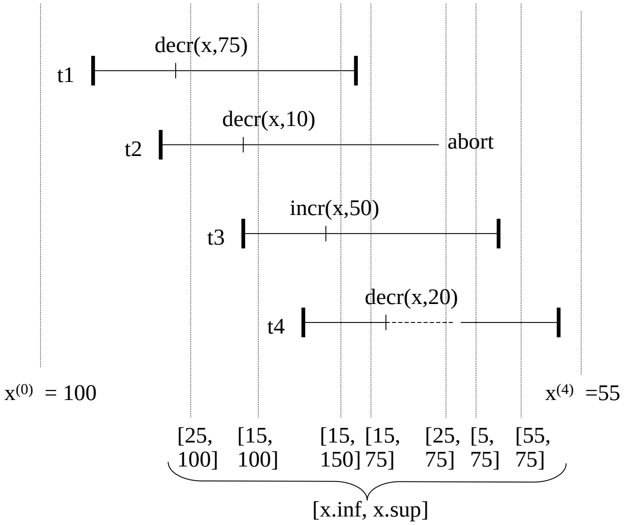

Figure 7.9: Example execution under Escrow locking.

is computationally more expensive than for simple wait-for graphs. However, given that Escrow locking just serves to reduce lock waits and deadlocks should be rare anyway, this additional complexity in deadlock handling is not a severe caveat.

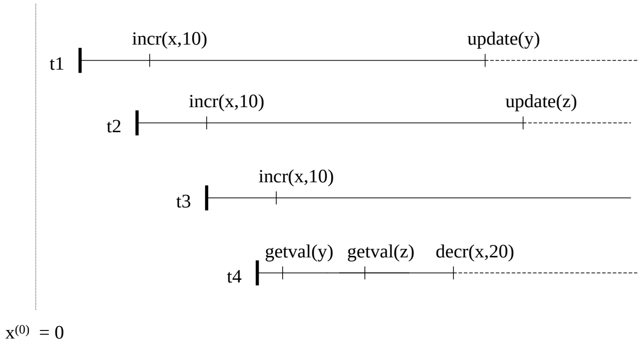

Figure 7.10: Example for a deadlock with Escrow locking.

tolerating bounded inconsistency

An interesting twist on exploiting state-based commutativity on counters is to relax the correctness requirements and tolerate some bounded inconsistency. More precisely, rather than guaranteeing that all reads of counter objects yield consistent values that are in line with a serializable history, an approximate value that falls into some form of "confidence interval" may be sufficient for the application. In contrast to a confidence interval used in statistics, however, the retrieved value should be guaranteed to be within the interval's bounds rather than having this property merely with very high probability.

In certain situations this behavior could be achieved without any concurrency control at all (other than short-term locks for the duration of a high-level operation), namely, when the Δ by which the counter value can be changed by a single transaction is very small and thus negligible relative to the absolute counter value.

In such a situation (e.g., when the counter is the total of a large number of individual counters and updates manipulate only the individual counters), the number of concurrently active transactions times the maximum Δ per individual transaction yields the maximum deviation from the actual value that could be observed when all transactions ran in a serializable manner.

ϵ-serializability

However, in other situations, this kind of confidence interval may be unacceptably large, and we would wish to have explicit control over the observable deviation from the actual value. A criterion along these lines is the notion of *ϵ-serializability* which assures that each read operation sees a value that deviates by at most ϵ from the value that it would see in a serial schedule. Obviously, the approach of tracking the infimum and supremum of a counter's value that we discussed for Escrow locking points us to possible implementation techniques for ϵ-serializability: basically, a read operation needs to be blocked when the difference between the supremum and the infimum is larger than the acceptable ϵ.

The ϵ-serializability approach can even be generalized to incorporate updates that are not necessarily commutative: as long as the value of the counter remains within a distance of ϵ from the value that would be produced in a serial schedule, the situation may be considered acceptable (depending on the application). This generalization requires additional mechanisms known as "divergence control" in the literature. An elegant property of the entire approach is that standard serializability is included as a special case, namely, for $\epsilon = 0$. In practice, a difficulty that could easily be underestimated is to understand the application good enough to derive a truly acceptable setting for ϵ. These are exactly the difficulties that are avoided by serializability (i.e., the case $\epsilon = 0$), which frees up application architects and developers even from thinking about the pitfalls of concurrency.

7.7 Lessons Learned

Based on the previous chapter's results on transaction forests with semantically rich operations, we have developed a couple of 2PL-based concurrency control protocols for the object model. In the simpler form, layered 2PL, we assume a layered structure of the transaction trees and simply apply 2PL at each layer. This is the fundamental concept in record-granularity locking that industrial-strength data servers support: record locks are held for transaction duration, and short-term page locks are held for the scope of record operations to ensure that the corresponding subtransactions can be isolated. In practice, the restricted nature of record operations may even allow us to further relax the subtransaction 2PL to the extreme point where pages merely need to be latched while being accessed. The latter technique can be viewed as very light-weight form of subtransaction locking.

The same principles carry over to other protocols, where 2PL is replaced by another conventional protocol at selected levels, as long as this protocol guarantees order-preserving conflict serializability. This way, optimistic or multiversion protocols can be integrated in a layered architecture, and even hybrid protocols are feasible with different concurrency control techniques at different levels.

For arbitrary transaction forests that do not conform to a layered structure, the protocol becomes more complex, as it needs to deal with situations where some transactions access certain data only through some ADT object interface whereas others access that data directly. The notion of retained locks allows us to generalize the key properties of the layered 2PL to such an architecture, but the need to remember low-level locks past the completion of subtransactions and the additional complexity of the lock conflict testing may incur noticeable overhead. So although the general object-model 2PL can cope with arbitrary collections of business objects, a performance-conscious system design will probably strive to limit the depth of the transaction forests or may even choose to embed the transaction forests into a layered structure.

Throughout most of this chapter, we have assumed a state-independent notion of commutativity. As a special case, when all direct children of all transaction roots are a priori known to be commutative (or more generally, some slice of operations across all transaction trees has this property), it is sufficient to provide concurrency control measures for the subtransactions that correspond to the roots' children. In such a setting, which is known as *decomposed transactions with chained subtransactions*, the subtransactions can be treated as if they were independent transactions as far as concurrency control is concerned. Exploiting this special case is, however, highly dependent on the application design. In a similar vein, special-purpose application objects may want to exploit even state-dependent commutativity. To this end, we have presented the efficient Escrow locking protocol that is specifically geared for objects of type counter with possible constraints on the counter value. This protocol can be integrated into the general object-model 2PL or the layered variant on a per-object basis.

Exercises

1. Investigate what kind of deadlocks can arise in a layered system with layered 2PL, and how they can be detected and eliminated. Under what conditions is it sufficient that each layer tests only for "local" deadlocks caused by lock waits of one layer only? Under what conditions is it necessary to consider lock waits of all layers "globally" to detect all deadlocks?

2. Discuss the pros and cons of the selective layered 2PL. As an example, consider

query-, record-, and page-level operations along the lines of Figure 7.4. Does it have performance advantages to have all three levels involved in locking rather than only two? If locking should be limited to two levels, are there arguments on which levels should be chosen?

3. Trace the lock requests, conflict tests, lock acquisitions, and lock releases for the execution of the schedule in Figure 7.7 under the general object-model 2PL with a SS2PL variant. How would the sample execution change if the append operations were dropped and t_2's decrement operation was replaced by an *incr*(a) operation that commutes with other increments? How would the execution change if t_2 started first and t_1 was spawned after all of t_2's operations but before the commit of t_2?

4. Develop pseudocode for a restricted version of the object-model 2PL, where a generalized form of 2-level schedules is considered with some transactions bypassing the object level L_1 and invoking L_0 operations directly. This setting is, for example, of interest in the context of federated information systems, where global transactions access a variety of databases through a federation layer, using commutativity-based 2PL for high-level operations exported by the underlying databases, but local transactions are still allowed to access a database directly. Such an architecture is sometimes called a federation of *"autonomous"* databases.

5. Apply the following hybrid protocols to the schedule of Figure 7.3 in a selective way so that only two levels are involved in concurrency control:

   (a) the forward-oriented optimistic concurrency control (FOCC, see Chapter 4) at the page level, and strong 2PL at the record level,

   (b) the forward-oriented optimistic concurrency control at the page level, and strong 2PL at the query level,

   (c) the hybrid multiversion protocol ROMV (see Chapter 5) at the page level, and strong 2PL at the record level,

   (d) the hybrid multiversion protocol ROMV at the page level, and strong 2PL at the query level.

   Sketch the resulting executions.

6. Prove the correctness of the 2PL protocol for 2-level schedules with deferred lock conflict testing based on return-value commutativity sketched in Section 7.6.

7. Trace the intermediate states of the Escrow data structure (i.e., the infimum and supremum values) for the counter object x in the example execution of Figure 7.9.

8. Consider two counter objects x and y, with initial values $x = 100$ and $y = 50$. Both counters have zero as a lower bound and no upper bound. Apply the Escrow locking method to the following schedule of three transactions one of which aborts:

   $decr_1(x, 60) incr_2(x, 20) incr_1(x, 10) decr_3(x, 50) decr_2(y, 60) incr_2(x, 20) a_2$
   $decr_1(y, 10) c_1 c_3$

9. Escrow operations can be further generalized into *conditional* increment and decrement operations, where the conditions may be stronger than the actual resource quantity needed, taking the form:

   conditional_decr (x, ϵ, Δ):
   `if x ≥ ε then x := x - Δ fi;`

   and an analogous conditional_incr operation. For the conditional_decr, the value of the ϵ parameter would be larger than or equal to the operation's Δ value, and the situation would be the other way round for the conditional_incr operation. In addition to such operation-specific conditions, the global invariants for `x.low` and `x.high` must be satisfied at all times. Discuss the notions of general and return-value commutativity for these generalized Escrow operations.

Bibliographical and Other Remarks

Layered 2PL has probably been around implicitly for record-level locking in commercial database systems for a long time. However, it has not been made explicit, identifying its principles rather than viewing it as an implementation trick, before the work of Weikum and Schek (1984) (see also Weikum (1991)) and Moss et al. (1986). The technique of selectively layered locking is discussed in the work of Weikum and Schek as well. The use of layered 2PL in the context of intra-transaction parallelism has been discussed by Weikum and Hasse (1993, 1997). Since some commercial database systems can parallelize update statements, these systems must use similar protocols, probably with clever latching instead of short-term page locks.

The special case of chained subtransactions, i.e., 2-level schedules where all L_1 operations generally commute, was coined "Sagas" by Salem and Garcia-Molina (1987) and "S-transactions" by Veijalainen (1989) and Veijalainen et al. (1992).

The decomposition of application processes into chains of "unbundled" steps that can be treated as independent transactions as far as concurrency control is concerned has been used in high-performance application design (e.g., for reservation systems) for a long time, but it has not been systematically rationalized before the above publications.

The general (i.e., non-layered) case of object-model locking has first been studied by Muth et al. (1993). Similar approaches have been discussed by Deacon et al. (1994) and Schek et al. (1991) for the more restricted setting of federated database systems with a global federation layer and a layer of local databases underneath which can also be accessed directly, bypassing the federation layer.

Layered concurrency control protocols that combine locking on one level with an optimistic or multiversion protocol on another layer are already discussed by Weikum (1991). An interesting hybrid protocol that exploits commutativity for update operations and versioning for read operations is discussed by Jagadish, Mumick, and Rabinovich (1997), with the special restriction that all update operations are assumed to be commutative among themselves (but do not commute with read operations). Escrow locking for counter objects has been developed by O'Neil (1986); the complexity of Escrow deadlocks is studied in O'Neil (1991), and generalizations of Escrow locking have been proposed by Härder (1988) as well as Soparkar and Silberschatz (1990). Preliminary forms of Escrow-style concurrency control can be traced back to Gawlick and Kinkade (1985) and Reuter (1982) and were used in the database and transaction-processing product IMS/VS Fast Path already in the 1970s. The notion of ϵ-serializability for bounding inconsistent reads has been developed by Wu et al. (1997) and fully formalized by Ramamritham and Pu (1995). Related approaches have been studied by Garcia-Molina and Wiederhold (1982) as well as Barbara and Garcia-Molina (1992).