Chapter 11

Transaction Recovery

"And if you find a new way, you can do it today. You can make it all true. And you can make it undo." (Cat Stevens)

11.1 Goal and Overview of the Chapter

With this chapter we begin the book's Part III, considering recovery methods to ensure the atomicity and persistence of transactions. The current chapter focuses on the issue of *transaction recovery*, which is needed to implement transaction *rollbacks* while the server continues to process other transactions. A rollback is necessary when a transaction is aborted either because of an unacceptable exception in the application program currently executing at the client site (e.g., a pointer error) or upon detecting inconsistencies in the server's data that would arise from the current transaction's further execution. In addition, the server may choose to abort a transaction for server-internal reasons such as deadlock or overload (see Chapters 4 and 10). In all these cases, which are commonly subsumed under the term "transaction failures", the server continues to operate and is expected to restrict its recovery measures to the actually affected transaction while processing other transactions as usual.

Thus, the issue in transaction recovery is not so much to provide fault-tolerant service, which is something we will discuss in subsequent chapters when we consider server failures. Rather, this chapter focuses on the problem of how to *undo* the effects of an aborted transaction in the presence of concurrent transactions. This problem is all but trivial; in particular, it is not avoided by serializability-enforcing measures alone, and it is not avoided by simply suspending all currently active transactions while an aborted transaction is being rolled back either. As a canonical example for

undo of aborted transaction

dirty-read anomaly

illustrating the problem, reconsider the *"dirty-read" anomaly* from Chapter 2:

$$w_1(x)r_2(x)c_2a_1$$

This history should be considered unacceptable, since t_2's effects depend on having read a "dirty" (i.e., not yet committed) update that t_1 has made earlier. Later on, when t_1 aborts, we would expect some form of "cascading abort" that includes the dependent transaction t_2 as well, but at this point it is too late to abort the already committed transaction t_2. On the other hand, a scheduler based on serializability would surely accept this history, since it is perfectly conflict-serializable. (Notice in this little discussion that an exchange of the two termination operations would immediately solve the problem, as the outcome of t_1 would be known at the end of t_2.)

In order to establish a class of acceptable histories that takes both serializability and transaction-recovery issues into account appropriately, we will first make the necessary actions for rolling back a transaction explicit in that we expand a transaction abort into actual data operations that undo the transaction's prior updates. These actions can be determined from a *log* of each transaction's updates that is kept at the server. As the server itself is not affected by a single transaction failure, this log can be maintained in the server's volatile memory. For the same reason, the exact implementation of the log does not really matter; we will later pay more attention to the design of its underlying data structure once we start considering server failures in Chapter 12. For the time being, we can think of logs as lists of page values prior to the corresponding page-write operations, organized on a per-transaction basis. More generally, a log can be perceived as a list of *inverse data operations*.

log of inverse operations

Once the previously implicit undo steps for an abort action are made explicit in what we will call an *expanded* schedule, we can again use serializability-based arguments along the lines of commutative operations for reasoning about the correctness of transaction aborts and their interdependencies with concurrent transactions. We will present the theory that builds upon this idea for the page model of transactions first: we will introduce a notion of reducibility that takes undo operations into account, then we will study a number of easy-to-implement, and syntactical, yet sufficient conditions for ensuring correctness, and we will finally look into the question of how to augment already known scheduling protocols in such a way that transaction aborts are correctly handled and anomalies like dirty reads are eliminated. Once the simpler case of page-model transactions is completely covered, we will consider the more advanced case of object-model transactions, and we will see that again some fundamental principles from the page model can be carried over, but need appropriate generalizations.

The chapter is organized as follows. Section 11.2 formalizes the concept of an expanded schedule. On this basis Sections 11.3, 11.4, and 11.5 discuss fundamental correctness criteria, sufficient conditions that are more easily implementable, and further relationships among the various schedule classes that are of theoretical interest. Section 11.6 then exploits the conceptual underpinnings by showing how page-model concurrency control methods can be easily extended to cope with transaction aborts. As the fundamental reasoning behind expanded schedules can be carried over to transaction executions with semantically rich operations and operation invocation trees, correctness criteria and algorithms for the object model can be discussed in a concise manner in Sections 11.7 and 11.8.

11.2 Expanded Schedules with Explicit Undo Operations

A major goal in this chapter is to derive a general correctness criterion for recovery that will be called *prefix-reducibility*, and to characterize the corresponding class of schedules in a way that indicates how to augment known scheduling protocols in this direction. To begin with, we give an overview of the approach and summarize the underlying ideas.

11.2.1 Intuition and Overview of Concepts

The essence behind our approach to treating recovery from a conceptual point of view is to make the actions that are associated with an abort operation occurring in a schedule or history explicit, and to treat them in the same way as ordinary operations. After a transaction failure, the goal is to restore the database to the state it was in before that transaction started; since this is not always possible in the presence of multiple transactions that are executing concurrently, let us consider it sufficient to undo the *write* operations the transaction to be aborted has executed. If we include corresponding operations in the transaction that is to be terminated abnormally, abort operations become obsolete, since what they need to accomplish is now explicit in the schedule or history in question.

As an example, consider the following history:

a history

$$s = r_1(x)w_1(x)r_2(x)a_1w_2(x)c_2$$

Obviously, we see a dirty-read situation here, since the second transaction reads a value of x produced by the first, but since transaction t_1 aborts, that value should not have been made available to transaction t_2. We choose to "implement" operation a_1 by writing back the original value of x preceding $w_1(x)$ (assuming that this value

can be obtained from the log); after that, we simply commit the transaction as if it had been successful. So now we are looking at the following history:

and its expansion

$$s' = r_1(x)w_1(x)r_2(x)w_1^{-1}(x)c_1w_2(x)c_2$$

We assume that the two writes of t_1 on x are inverse to each other, so that the second compensates for the first. Thus, t_1 has no effect on the database, and now the problem is a different one: Since s' has a cyclic conflict, it is an ordinary scheduling problem to avoid non-serializability, which can be accomplished by any of the known protocols. Notice that a 2PL scheduler, for example, would allow the original history s, as it is perfectly conflict-serializable, but would detect the unacceptable behavior in the expanded history s'. Thus, if we make sure that s is replaced by s' prior to execution, we are guaranteed that no unacceptable history will result.

11.2.2 The Formal Model

We now make the idea of expanding histories in response to a request to abort a transaction precise, and we will show that expanded histories give rise to a uniform correctness criterion. As was briefly mentioned already, expanding histories is essentially based on the following concepts:

idea of expansions

1. All actions related to failures are made explicit in a history, i.e., are added to a given history as additional steps, and any abort operation is replaced by a commit operation;
2. a history so expanded is treated by ordinary serializability arguments, where special attention is paid to the interaction between steps on behalf of transaction aborts and regular steps.

Since we assume log-based recovery, we may assume that a transaction that shall be aborted and hence needs its effects made undone can simply be compensated for by "inverse" write operations: If s is a history s.t. $t_i \in \text{abort}(s)$ and $w_i(x) \in \text{op}(s)$, then the undo operation corresponding to $w_i(x)$ is yet another write operation on x which restores the original value of x; such an operation is denoted by $w_i^{-1}(x)$ in what follows.

Read operations, on the other hand, will not need special treatment when aborting a transaction; we can assume that the results of a read are simply "forgotten", so that the steps themselves can be considered as "null steps."

Now each undo operation of the form $w_i^{-1}(x)$ undoes a previously executed operation of the form $w_i(x)$ of transaction t_i, so that it makes sense to put undo operations in *reverse order* of the corresponding "do" operations in a history. If an aborted transaction has been expanded in this way, it can be treated like a committed one from that point on, so that we arrive at the following basic expansion strategy: If $t_i \in \text{abort}(s)$ holds in a given history s, i.e., $a_i \in \text{op}(s)$, then a_i is replaced by the undo operations of all write steps of t_i (in reverse order), followed by c_i.

Example 11.1

Consider history

$$s = r_1(x)w_1(x)r_2(y)w_1(y)w_2(y)a_1r_2(z)w_2(z)c_2.$$

replacing aborts by inverse writes

According to what was just said, s is rewritten into the following history (which will be called the *expansion* of s or the *expanded history* for s shortly):

$$s' = r_1(x)w_1(x)r_2(y)w_1(y)w_2(y)w_1^{-1}(y)w_1^{-1}(x)c_1r_2(z)w_2(z)c_2$$

□

So far we have considered histories, and of course we will need an extension of our approach to prefixes of histories, i.e., to schedules. Indeed, in a running scheduling situation a scheduler commonly sees prefixes of histories only; if there is no explicit abort operation yet, it is not immediately clear how to place abort actions in a schedule for a transaction that is still active.

To this end, we now make the following assumption: If s is a schedule and not a history, i.e., for some transactions in s it is still open how they will end (i.e., $\text{active}(s) \neq \emptyset$), s will be treated as if a system failure had occurred after the last operation in s so far. In other words, all transactions $t_i \in \text{active}(s)$ will be (implicitly) aborted and then of course be treated in the way described above. The only aspect that is different is the fact that now the reverse ordering of operations is no longer local to a single transaction, but global w.r.t. all aborted transactions.

Example 11.2

Consider schedule

$$s = w_1(x)w_2(x)w_2(y)w_1(y).$$

terminating active transactions

Since $t_1, t_2 \in \text{active}(s)$, s still appears acceptable, although there is already a cyclic conflict between t_1 and t_2. If both transactions are aborted according to the assumption above, we must be careful in sequencing the corresponding abort operations. Indeed, neither the ordering a_1a_2, resulting in

$$s' = w_1(x)w_2(x)w_2(y)w_1(y)w_1^{-1}(y)w_1^{-1}(x)c_1w_2^{-1}(y)w_2^{-1}(x)c_2$$

nor the ordering a_2a_1, resulting in

$$s'' = w_1(x)w_2(x)w_2(y)w_1(y)w_2^{-1}(y)w_2^{-1}(x)c_2w_1^{-1}(y)w_1^{-1}(x)c_1$$

represents a correct execution. Instead, a correct undo can in this case only be guaranteed if the undos corresponding to different aborts are executed in an overlapping fashion; Thus, the appropriate expansion of s must look as follows:

$$s''' = w_1(x)w_2(x)w_2(y)w_1(y)w_1^{-1}(y)w_2^{-1}(y)w_2^{-1}(x)w_1^{-1}(x)c_2c_1$$

□

We could formally capture situations like the one just seen through the notion of a *group abort*, representing the collective abort actions of all active transactions. However, we can do without explicit group aborts as long as we keep in mind that active transactions need a slightly different treatment than aborted ones.

We are now ready to make the above considerations precise as follows:

expansion of a schedule

Definition 11.3 (Expansion of a schedule)

Let s be a schedule. The *expansion* of s, denoted $\exp(s)$, is defined as follows:

(i) Steps of $\exp(s)$:

(a) $t_i \in \text{commit}(s) \Rightarrow \text{op}(t_i) \subseteq \text{op}(\exp(s))$

(b) $t_i \in \text{abort}(s) \Rightarrow (\text{op}(t_i) - \{a_i\}) \cup \{c_i\} \cup \{w_i^{-1}(x) \mid w_i(x) \in t_i\} \subseteq \text{op}(\exp(s))$

(c) $t_i \in \text{active}(s) \Rightarrow \text{op}(t_i) \cup \{c_i\} \cup \{w_i^{-1}(x) \mid w_i(x) \in t_i\} \subseteq \text{op}(\exp(s))$

(ii) Step ordering in $\exp(s)$:

(a) All steps from $\text{op}(s) \cap \text{op}(\exp(s))$ occur in $\exp(s)$ in the same order as in s;

(b) all inverse steps of an aborted transaction occur in $\exp(s)$ after their original steps and before the respective commit operation;

(c) all inverse steps of transactions in $\text{active}(s)$ occur in $\exp(s)$ after the original steps of s and before there corresponding commits;

(d) the ordering of inverse steps is the reverse of the ordering of the corresponding original steps.

Example 11.4

Consider schedule

$$s = r_1(x)w_1(x)r_2(y)w_2(y)r_3(z)w_3(z)r_4(y)w_4(y)a_1c_3r_2(z)w_2(z).$$

Then we have:

$$\exp(s) = r_1(x)w_1(x)r_2(y)w_2(y)r_3(z)w_3(z)r_4(y)w_4(y)$$
$$w_1^{-1}(x)c_1c_3r_2(z)w_2(z)w_2^{-1}(z)w_4^{-1}(y)w_2^{-1}(y)c_2c_4.$$

□

11.3 Correctness Criteria for the Page Model

Now that we have fixed a way in which we can treat aborted transactions in such a way that serializability-based schedulers could be enabled to discover undesirable situations that stem from transaction failures, we need to cast this approach into a formally precise correctness criterion. To this end, we will present a series of correctness criteria which, as will be seen, are in a sense orthogonal to each other. For practical purposes, we will later choose the one that exhibits the most attractive properties.

11.3.1 Expanded Conflict Serializability

As indicated above already, the notion of serializability can be extended to expanded histories and schedules *without* further restrictions or constraints. In particular, our well-known notion of conflict remains the same as before: Two steps from distinct transactions are *in conflict* if they access the same data item and at least one of them is a (direct or inverse) write operation.

expanded conflict serializability

Definition 11.5 (Expanded conflict serializability)

Let s be a history. s is *expanded conflict-serializable* if its expansion $\exp(s)$ is conflict-serializable.

Let XCSR denote the class of all expanded conflict-serializable histories.

Example 11.6

Consider

$$s = r_1(x)w_1(x)r_2(x)a_1c_2$$

Then we have

$$\exp(s) = r_1(x)w_1(x)r_2(x)w_1^{-1}(x)c_1c_2.$$

Since $\exp(s)$ exhibits a cyclic conflict between its two transactions, it follows that $s \notin$ XCSR. Next consider

$$s' = r_1(x)w_1(x)a_1r_2(x)c_2$$

Now we find

$$\exp(s') = r_1(x)w_1(x)w_1^{-1}(x)c_1r_2(x)c_2.$$

Since $\exp(s')$ is equivalent to the serial execution t_1t_2, it follows that $s' \in$ XCSR. □

Since the inverse write operations occurring in an expanded history can at most *add* to the conflict relation of a given history, it should be clear that if we omit them, we obtain a sub-history of the expanded history; if the expansion is conflict-serializable, so is its sub-history. Thus we find that class XCSR is a subclass of class CSR, and the first schedule in the previous example even shows that this inclusion is strict:

Lemma 11.7

XCSR ⊂ CSR

While expanded conflict-serializability is already a reasonable step towards a formal treatment of transaction failures, it occurs too restrictive in certain situations. The reason is that some intuitively correct histories do not belong to XCSR, as shown in the following example:

problems with XCSR

Example 11.8

Suppose all transactions in a given history are aborted, as in

$$s = w_1(x)w_2(x)a_2a_1.$$

For this history, we find

$$\exp(s) = w_1(x)w_2(x)w_2^{-1}(x)c_2w_1^{-1}(x)c_1.$$

which apparently has a cyclic conflict; thus, s is not in XCSR. On the other hand, the cycle is technically irrelevant, as both transactions are aborted. In that sense, s is acceptable. □

In Example 11.8, the acceptability of s crucially depends on the fact that the undo operation of t_2 immediately follows its corresponding "do operation", so that in principle we can remove this pair of operations from s without changing its effect. Note that this would no longer be the case if the two abort steps of s were in reverse order.

If we perform the removal, we obtain

$$w_1(x)c_2w_1^{-1}(x)c_1.$$

Since no operation from t_2 is left, we can drop c_2. Now $w_1(x)$ and its inverse operation have become adjacent, and by the same argument we just applied to t_2 we can now eliminate t_1 from s. So we are left with an empty schedule that does not perform any actions at all, which is exactly the desired effect, as both transactions were aborted.

We will now study this observation further and establish another correctness criterion that will ultimately serve as the goal of scheduling in what follows.

11.3.2 Reducibility and Prefix-Reducibility

If a write operation and its inverse happen to be "adjacent," i.e., they occur right after one another, it is guaranteed that no operation can be in between the two that is in conflict with either of them. If a write operation and its inverse are not adjacent, it may be possible to make them adjacent by performing allowed commutations in the given history, i.e., by moving them together through pairwise exchanges with other operations. We will next cast this intuition into *transformation rules* that allow us to re-order and modify a given history in certain ways. Note that the notion of expanded serializability as introduced above does not exploit information about the presence of inverse operations, or about their adjacency.

transformation rules

reducibility

Definition 11.9 (Reducibility)

A history s is *reducible* if its expansion $\exp(s)$ can be transformed into a serial history by finitely many applications of the following rules (in any order):

(i) Commutativity Rule (CR):
If $p, q \in \text{op}(\exp(s))$ s.t. $p < q$ and $(p, q) \notin \text{conf}(\exp(s))$, and if there exists no step $o \in \text{op}(\exp(s))$ s.t. $p < o < q$, then the order of p and q can be reversed.

(ii) Undo Rule (UR):
If $p, q \in \text{op}(\exp(s))$ are inverses of each other (i.e., of the form $p = w_i(x)$, $q = w_i^{-1}(x)$) with $p < q$) and if there is no step $o \in \text{op}(\exp(s))$ in between, i.e., $p < o < q$, then the pair p, q of steps can be removed from $\exp(s)$.

(iii) Null Rule (NR):
If $p \in \text{op}(\exp(s))$ has the form $p = r_i(x)$ s.t. $t_i \in \text{active}(s) \cup \text{abort}(s)$, then p can be removed from $\exp(s)$.

(iv) Ordering Rule (OR):
Two commutative, unordered operations can be arbitrarily ordered.

Let RED denote the class of all reducible histories.

Notice that the presence of UR, the Undo Rule, will in most cases have the effect that the original history is not just transformed into a serial history (if possible), but that it will indeed be reduced to something equivalent (and serial, if possible). Ultimately, a given reducible history may be reduced to a sequence of commit operations only, which is trivially serial (and also considered to be an "empty" schedule).

Example 11.10

Consider

$$s = r_1(x)w_1(x)r_2(x)w_2(x)a_2a_1$$

Then we have

$$\exp(s) = r_1(x)w_1(x)r_2(x)w_2(x)w_2^{-1}(x)c_2w_1^{-1}(x)c_1.$$

By using rules CR, UR, and NR, $\exp(s)$ can be reduced as follows:

by UR: $r_1(x)w_1(x)r_2(x)c_2w_1^{-1}(x)c_1$
by NR: $r_1(x)w_1(x)c_2w_1^{-1}(x)c_1$
by NR: $w_1(x)c_2w_1^{-1}(x)c_1$
by CR: $w_1(x)w_1^{-1}(x)c_2c_1$
by UR: c_2c_1

As a result, $s \in$ RED.

Notice that conflicts are defined for data operations only, so commit operations can freely be commuted with reads or writes as long as the ordering of steps given by a transaction is not modified. □

Example 11.11

Consider

$$s = w_1(x)r_2(x)c_1c_2$$

Here, by rule CR, the only relevant transformation is into

$$w_1(x)c_1r_2(x)c_2,$$

which is serial, so $s \in$ RED. Notice, however, that in this case the "reduction" of s has not really removed any operation. □

The notion of reducibility thus not only respects the commutativity of operations that are not in conflict in a given history or schedule, but also the elimination of adjacent do and undo operations.

As we have done for ordinary histories when we defined conflict serializability, we need to be able to decide about the reducibility of a given history by looking at a prefix only. As before, the reason for this is that a scheduler must be able to decide on the executability of a given operation *without* having seen the entire transaction or even the history under construction.. In particular, the scheduler must take the fact into account that a transaction can be aborted at any moment.

Example 11.12

Consider the history

$$s = w_1(x)w_2(x)c_2c_1.$$

Suppose a scheduler has already executed the two write operations, and then commits t_2. If at that point t_1 would abort, the resulting schedule (i.e., prefix of s) would be expanded into

$$s' = w_1(x)w_2(x)c_2w_1^{-1}(x)c_1.$$

which is neither in XCSR nor in RED. The latter holds since none of the reduction rules applies (except for ones involving the commit operations, which does not help here), so reduction leaves the schedule essentially unchanged. Thus, s should not be acceptable, since it has an "unsafe" prefix. □

We are therefore interested in the closedness of expandibility as well as of reducibility under taking prefixes, which is defined next:

prefix-expanded conflict serializability

Definition 11.13 (Prefix-Expanded Conflict Serializability)

A history s is *prefix-expanded conflict-serializable* if each of its prefixes is expanded conflict-serializable.

Let PXCSR denote the class of all prefix-expanded conflict-serializable histories.

prefix-reducibility

Definition 11.14 (Prefix-Reducibility)

A history s is *prefix-reducible* if each of its prefixes is reducible.

Let PRED denote the class of all prefix-reducible histories.

Lemma 11.15

(i) PXCSR $\subset$ XCSR

(ii) PRED $\subset$ RED

Proof. Since every history is trivially a prefix of itself, both inclusions are straightforward. Next consider $s = w_1(x)w_2(x)c_2c_1$ from Example 11.12 once more. Obviously, $s \in$ RED–PRED; in particular, prefix $w_1(x)w_2(x)c_2$ of s is not reducible. This shows that inclusion (ii) is strict. Moreover, $s \in$ XCSR–PXCSR, witnessed by the same prefix; in other words, inclusion (i) is also strict. □

Example 11.16

Consider

$$s = w_1(x)w_2(x)a_2a_1$$

As we have discussed in Example 11.8 already, $s \notin$ XCSR; hence $s \notin$ PXCSR due to the previous lemma.

On the other hand, $s \in$ RED, since our discussion following Example 11.8 has already revealed that s can be reduced into a serial schedule. Moreover, it is easily verified that every prefix of s is also reducible. Thus, $s \in$ PRED. □

To conclude this section, we establish the following relationships between the classes of histories introduced so far:

Theorem 11.17

(i) XCSR $\subset$ RED

(ii) PXCSR $\subset$ PRED

(iii) XCSR and PRED are incomparable w.r.t. set inclusion.

Proof. (i) Consider some $s \in$ XCSR. By definition, $\exp(s)$ is thus conflict-serializable, so there exists a serial history s' which is conflict-equivalent to $\exp(s)$. Clearly, $\exp(s)$ can be transformed into s' using the rules from Definition 11.9 (except for UR and NR), performing allowed commutations or ordering previously unordered steps only. Thus, $s \in$ RED. History s from Example 11.8 shows that the inclusion is strict.

(ii) Suppose that $s \in$ PXCSR. Then each prefix s' of s is in XCSR and hence in RED; thus, $s \in$ PRED. Again, history s from Example 11.8 shows that the inclusion is strict.

(iii) In order to show that XCSR and PRED are incomparable w.r.t. set inclusion, it suffices to consider the following histories:
$w_1(x)w_2(x)c_2c_1 \in$ XCSR–PRED (Example 11.12)
$w_1(x)w_2(x)a_2a_1 \in$ PRED–XCSR (Example 11.8) □

Part (iii) of the previous theorem states that reducibility is independent from the considerations on prefixes we have done above. We will later prove another theorem which states that PRED contains all intuitively correct histories.

11.4 Sufficient Syntactic Conditions

Consider again the history we looked at in the beginning of this chapter:

dirty-read anomaly

$$w_1(x)r_2(x)c_2a_1$$

We have already discussed why this history is not acceptable, and that a serializability-based scheduler would nevertheless consider it valid. In this section we study correctness criteria for transaction recovery that are exclusively based on simple syntactical restrictions on the allowed histories. Essentially, these restrictions state in what relative order the termination operations from various transactions that have interdependencies (such as read-from situations) are allowed to occur. And as will be seen, putting more and more restrictions on the allowed placement of termination operations or on the relative ordering of reads, writes, aborts, and commits increasingly simplifies transaction recovery, but also limits the amount of parallelism that can be achieved. Compared to the criteria we have introduced earlier in this chapter, which emphasize a way of *processing* aborts, the syntactic notions presented next can be seen as sufficient conditions for correct recovery, as will become clear when we study the relationships that exist between the various notions.

11.4.1 Recoverability

Our first syntactic notion is intended to avoid dirty read situations which may arise when one transaction has read a data object from another and then commits "too early".

recoverability

Definition 11.18 (Recoverability)

A history s is *recoverable* if the following holds for all transactions $t_i, t_j \in \text{trans}(s)$, $i \neq j$: If t_i reads from t_j in s and $c_i \in op(s)$, then $c_j <_s c_i$.
Let RC denote the class of all recoverable histories.

Thus, in a recoverable history each transaction can only be committed after all transactions from which it has read have been committed.

Example 11.19

Consider

$$s_1 = w_1(x)w_1(y)r_2(u)w_2(x)r_2(y)w_2(y)w_3(u)c_3c_2w_1(z)c_1$$

Then t_2 reads y from t_1 and $c_2 \in s$, but $c_1 \not<_s c_2$. It follows that $s_1 \notin$ RC. Next consider

$$s_2 = w_1(x)w_1(y)r_2(u)w_2(x)r_2(y)w_2(y)w_3(u)c_3w_1(z)c_1c_2$$

Then $s_2 \in$ RC, since the commit operation of t_2 follows after that of t_1. □

11.4.2 Avoiding Cascading Aborts

In history s_2 in the previous example we can identify another problem: If t_1 would be aborted right after operation $r_2(y)$, we would again encounter a dirty-read situation, and the abort of t_1 would imply the abort of transaction t_2. In other words, sometimes aborts "cascade", and we can avoid this by a suitable restriction of class RC:

borts

Definition 11.20 (Avoiding cascading aborts)

A history s *avoids cascading abort* if the following holds for all transactions $t_i, t_j \in \text{trans}(s)$, $i \neq j$: If t_i reads x from t_j in s, then $c_j <_s r_i(x)$.
Let ACA denote the class of all histories that avoid cascading aborts.

Note that the cascading abort in the above example (s_2) is an inconvenience, but unlike the canonical dirty-read anomaly considered before, it does not incur any correctness problems as the transaction on which the dirty reader depends is not yet committed at the time the first would abort.

Thus, while recoverability of a history means that each transaction reaches its commit point *after* all transactions from which it has read are committed, the ACA property means that a transaction can only read values written by transactions that are already committed, i.e., whose presence in the database is, in a sense, "guaranteed."

Example 11.21

For history s_2 from Example 11.19 we thus have $s_2 \notin$ ACA. However, for history

$$s_3 = w_1(x)w_1(y)r_2(u)w_2(x)w_1(z)c_1r_2(y)w_2(y)w_3(u)c_3c_2$$

we can verify that $s_3 \in$ ACA. □

History s_3 is still not entirely free of problems: Suppose that t_1 aborts right after $w_2(x)$; then there is no need to abort t_2 as well, but the database is brought back into the state it was in prior to the beginning of t_1, although t_2 has already written a new value of data item x. As we have mentioned earlier, the by far most practical approach to implement transaction rollbacks is by undoing operations of aborted transactions. In contrast, some histories allowed under RC or ACA are intuitively recoverable only by first going back to a potentially very old state and then redoing a potentially long sequence of committed transactions. Obviously, such an approach is inadequately expensive for merely rolling back a single transaction while the server is continuing to process other transactions.

Example 11.22

Consider a numeric data item x with an initial value of $x = 1$, and let history $s = w_1(x,2)w_2(x,3)c_2a_1$ be given, where $w_i(x,n)$ means that w_i writes value n for x. Thus, the undo of $w_1(x,2)$ would restore the value as it was immediately before the operation was executed, setting x to 1; however, after a_1 the current value of x should be 3. So in this example, the only correct form of transaction recovery for t_1 would be to restore the value as of the beginning of the entire schedule and then redoing the write of the already committed transaction t_2. □

The general problem here is that the values that have to be restored after a transaction abort can differ from those that immediately preceded the write operations of the aborted transaction. Such differences generally result from mutual overwrites performed by active transactions whose outcome is open at the time they write.

11.4.3 Strictness

To avoid being forced to perform a more expensive transaction recovery different from the simple undo approach, we impose yet another syntactical restriction on histories.

strictness

Definition 11.23 (Strictness)

A history s is *strict* if the following holds for all transactions $t_i \in \text{trans}(s)$ and for all $p_i(x) \in \text{op}(t_i)$, $p \in \{r, w\}$: If $w_j(x) <_s p_i(x)$, $i \neq j$, then $a_j <_s p_i(x) \vee c_j <_s p_i(x)$.
Let ST denote the class of all strict histories.

In words, a history is strict if no data item is read or overwritten until the transaction that wrote it last has ended (either by commit or by abort).

Example 11.24

For history s_3 from Example 11.21 we find $s_3 \notin$ ST. Next, for history

$$s_4 = w_1(x)w_1(y)r_2(u)w_1(z)c_1w_2(x)r_2(y)w_2(y)w_3(u)c_3c_2$$

we can verify that $s_4 \in$ ST. □

11.4.4 Rigorousness

Our next class of recoverable histories is a restriction of ST that is based on the following intuition: A history that is in class ST informally avoids write-read as well as write-write conflicts between uncommitted transactions; a history in the class RG defined next completes that picture by additionally avoiding read-write conflicts between uncommitted transactions. As will be seen shortly, this extra provision makes a remarkable difference in the relationship to serializability.

rigorousness

Definition 11.25 (Rigorousness)

A history s is *rigorous* if it is strict and additionally satisfies the following condition: For all transactions $t_i, t_j \in \text{trans}(s)$, if $r_j(x) <_s w_i(x), i \neq j$, then $a_j <_s w_i(x) \vee c_j <_s w_i(x)$.
Let RG denote the class of all rigorous histories.

In words, a history is rigorous if it is strict and no object x is overwritten until all transactions that read x last are finished.

Example 11.26

For history s_4 from Example 11.24 we find $s_4 \notin$ RG. Next, for history

$$s_5 = w_1(x)w_1(y)r_2(u)w_1(z)c_1w_2(x)r_2(y)w_2(y)c_2w_3(u)c_3$$

we can verify that $s_5 \in$ RG. □

The relationship between the syntactical classes of histories is intuitively such that they continuously put further restrictions on the set of admissible schedules, as suggested by our sample histories s_1–s_5. More formally, we can show the following theorem:

Theorem 11.27

RG ⊂ ST ⊂ ACA ⊂ RC

Proof. The fact that RG ⊆ ST immediately follows from the definition of RG. That the inclusion is strict is verified by history s_4 from Example 11.26 ($s_4 \in$ ST–RG).

Next let $s \in$ ST, and let t_i read x from t_j in s, $i \neq j$. By the definition of the reads-from relation, $w_j(x) <_s r_i(x)$ and $a_j \not<_s r_i(x)$ then hold. By the definition of class ST, this implies $c_j <_s r_i(x)$, i.e., $s \in$ ACA. Thus, ST ⊆ ACA; history s_3 from Example 11.21 proves that the inclusion is strict ($s_3 \in$ ACA − ST).

Next consider some history $s \in$ ACA, and suppose that t_i reads x from t_j in s, $i \neq j$, and that $c_i \in s$. Since $s \in$ ACA, it follows that $c_j <_s r_i(x)$. Clearly, we also have $w_j(x) <_s c_j$ and $r_i(x) <_s c_i$. This implies $c_j <_s c_i$, i.e., $s \in$ RC. Thus, ACA ⊆ RC; history s_2 from Example 11.19 moreover shows that this inclusion is also strict ($s_2 \in$ RC−ACA). □

Another property that is easily verified by the definitions of RG, ST, ACA, and RC is that membership in either of the four classes is prefix-commit closed, i.e., if a

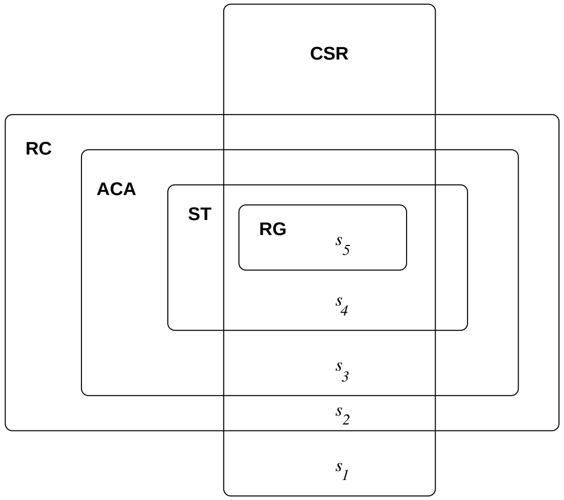

Figure 11.1: Relationship between history classes RC, ACA, ST, RG, and CSR.

history is, say, in ACA, the same holds for the committed projection of each of its prefixes (see Exercise 4).

We next look at the relationship between our syntactic classes of histories and the serializable ones, where we restrict the attention to the most important class of conflict-serializable schedules. Surprisingly, this relationship essentially is such that ST, ACA, and RC are incomparable to CSR w.r.t. set inclusion, while RG is not. This is illustrated in Figure 11.1, where the various histories mentioned are those from Examples 11.19, 11.21, 11.24, and 11.26, resp. Moreover, history

$$w_1(x)w_2(x)w_2(y)c_2w_1(y)c_1$$

is an example of a history that is in RC, but not in CSR. We mention that Figure 11.1 can be extended to also include the classes of commit-view-serializable (CMVSR) and commit-final-state-serializable (CMFSR) schedules (see Chapter 3 for these classes).

While the incomparability between CSR and classes ST, ACA, and RC w.r.t. set inclusion is easily established through examples, containment of RG in CSR is

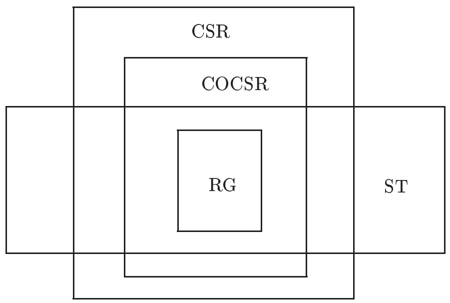

Figure 11.2: Relationship between history classes CSR, COCSR, RG, and ST.

implied by the following observation regarding RG and COCSR, the subclass of CSR containing all commit-ordered conflict-serializable histories: Membership of a history s in class RG actually means that for any two operations $p \in \text{op}(t_i)$ and $q \in \text{op}(t_j)$, $i \neq j$, that are in conflict in s such that $p <_s q$, commit operation c_i also comes before q in s (provided that c_i indeed occurs in s). On the other hand, since q precedes c_j (if c_j occurs in s), we may conclude that c_i precedes c_j. Thus, we have shown that RG $\subseteq$ COCSR History $s = r_1(x)w_2(x)c_1c_2 \in$ COCSR–RG additionally shows that this inclusion is strict, so we have:

Theorem 11.28

RG $\subset$ COCSR

Figure 11.2 summarizes the relationships just described.

11.4.5 Log-Recoverability

Our next goal is to define a correctness criterion which is based on recoverability and in addition places an order on the termination operations of transactions that are involved in a write-write conflict. As will turn out, this new criterion, if combined with serializability, can serve as a characterization of class PRED.

Definition 11.29 (Log-recoverability)

A schedule s is *log-recoverable* if the following two properties hold:

(i) s is recoverable, i.e., for all transactions $t_i, t_j \in \text{trans}(s)$ $(i \neq j)$ we have: if t_i reads from t_j in s and $c_i \in s$, then $c_j <_s c_i$,

(ii) for all transactions $t_i, t_j \in \text{trans}(s)$ $(i \neq j)$ we additionally have: if there is a write-write conflict of the form $w_i(x) < w_j(x)$ in s, then $a_i < w_j(x)$ or $c_i < c_j$ if t_j commits, and $a_j < a_i$ if t_i aborts.

Let LRC denote the class of all log-recoverable histories.

In essence, the definition of LRC is about the correct placement of termination operations in such a way that an isolated undoing of transactions in conflict can be guaranteed. We illustrate the basic effect on two prototypical schedules next. Let us first look at

$$s = w_1(x)r_2(x)$$

Here, the first part of the definition of class LRC applies, and as the following table shows, the cases that are not log-recoverable are also not in PRED:

Case	History	Property
(1)	$w_1(x)r_2(x)a_1a_2$	$\in$ PRED
(2)	$w_1(x)r_2(x)a_1c_2$	$\notin$ PRED
(3)	$w_1(x)r_2(x)c_2c_1$	$\notin$ PRED
(4)	$w_1(x)r_2(x)c_2a_1$	$\notin$ PRED
(5)	$w_1(x)r_2(x)a_2a_1$	$\in$ PRED
(6)	$w_1(x)r_2(x)a_2c_1$	$\in$ PRED
(7)	$w_1(x)r_2(x)c_1c_2$	$\in$ PRED
(8)	$w_1(x)r_2(x)c_1a_2$	$\in$ PRED

Next, we look at schedule

$$s = w_1(x)w_2(x)$$

to which the second part of the definition of class LRC applies. Interestingly, the unacceptable termination possibilities are again exactly those which are not in PRED:

Case	History	Property
(1)	$w_1(x)w_2(x)a_1a_2$	$\notin$ PRED
(2)	$w_1(x)w_2(x)a_1c_2$	$\notin$ PRED
(3)	$w_1(x)w_2(x)c_2c_1$	$\notin$ PRED
(4)	$w_1(x)w_2(x)c_2a_1$	$\notin$ PRED
(5)	$w_1(x)w_2(x)a_2a_1$	$\in$ PRED
(6)	$w_1(x)w_2(x)a_2c_1$	$\in$ PRED
(7)	$w_1(x)w_2(x)c_1c_2$	$\in$ PRED
(8)	$w_1(x)w_2(x)c_1a_2$	$\in$ PRED

The following theorem shows that the above observations are not just a coincidence:

characterization of class PRED

Theorem 11.30

A schedule s is prefix-reducible iff it is conflict-serializable and log-recoverable, i.e., PRED = CSR $\cap$ LRC.

For being able to prove this theorem we need the following lemma:

Lemma 11.31

If $s \in$ LRC, then all operations of uncommitted transactions can be eliminated from exp(s) using rules CR, UR, NR, and OR.

Proof. The claim is trivially satisfied for read operations, since read operations of uncommitted transactions can always be dropped from exp(s) for any s. Therefore, we only need to prove the lemma for write operations of uncommitted transactions in s. Let transaction t_i be uncommitted in s s.t. $w_i(x) \in$ op(s). Assume that n operations are performed on x and preceding $w_i^{-1}(x)$ in exp(s). We will show by induction on n that $w_i(x)$ can be eliminated from some modification of exp(s).

First observe that for case $n = 0$, a direct application of rule UR eliminates $w_i(x)$ from exp(s).

Next, if $w_i(x)$ and $w_j(x)$ are such that $w_i(x) <_s w_j(x)$ and $a_i \not<_s w_j(x)$, we will say that $w_i(x)$ and $w_j(x)$ are in *active* conflict. Observe that if $w_i(x)$ and $w_j(x)$ are *not* in active conflict, but $w_i(x) <_s w_j(x)$, then in exp(s) we have $w_i^{-1}(x) < w_j(x)$.

Now for the induction step, let $n > 0$, and assume the lemma holds for all $k < n$. Assume that $o_j(x)$ is the last operation w.r.t. to the ordering in exp(s) among the n operations satisfying $w_i(x) < o_j(x) < w_i^{-1}(x)$ in exp(s). If $i = j$, then $o_j(x) = r_i(x)$, since each transaction contains at most one write operation per data object. But then $r_i(x)$ can be dropped from exp(s), leaving us with $n - 1$ operations between

$w_i(x)$ and its inverse, and by the induction hypothesis we conclude that $w_i(x)$ can be eliminated from exp(s). If $i \neq j$, we consider three cases:

(i) $o_j = r_j$: If transaction t_j is committed in s, then $s \notin$ LRC, a contradiction to our general assumption. If t_j is not committed, then r_j can be eliminated from exp(s) using rule NR, and by the induction hypothesis our claim holds.

(ii) $o_j = w_j$: If $w_i(x)$ and $w_j(x)$ are in active conflict, transaction t_j cannot be committed in s, since otherwise s would not be in LRC. Thus, t_j is either active or aborted in s, and in both cases exp(s) must have $w_i(x) < w_j(x) < w_j^{-1}(x) < w_i^{-1}(x)$. Thus, $w_j(x)$ is not the last operation on x w.r.t. the ordering in exp(s). Now if $w_i(x)$ and $w_j(x)$ are not in active conflict, then in exp(s) we have $w_i(x) < w_i^{-1}(x) < w_j(x)$, which contradicts our assumption.

(iii) $o_j = w_j^{-1}$: If $w_i(x)$ and $w_j(x)$ are in active conflict in s, we have $w_j^{-1}(x) < w_i^{-1}(x)$ in exp(s), and hence $w_i(x) < w_j(x)$. Operation $w_j(x)$ together with its inverse $w_j^{-1}(x)$ can be eliminated from exp(s) using the inductive assumption, as can be operation $w_i(x)$. If $w_i(x)$ and $w_j(x)$ are not in active conflict, i.e., t_i aborts before operation $w_j(x)$ in s, then we have $w_i^{-1}(x) < w_j(x)$ in exp(s). This again contradicts our assumption that $w_j^{-1}(x)$ is the last operation on x before $w_i^{-1}(x)$. $\square$

Proof of Theorem 11.30. "$\supseteq$:" Let $s \in$ CSR $\cap$ LRC, and assume that $s \notin$ PRED. Then there exists a prefix s' of s that is not reducible. From the definition of LRC it follows that s' is also in LRC; since membership in CSR is prefix-closed, s' is also in CSR. By Lemma 11.31, all operations of uncommitted transactions can be eliminated from exp(s'); call the result of this elimination s''. Thus s'' contains the committed transactions of s' only. Since $s' \in$ CSR, s'' can be transformed into a serial history by a finite number of applications of commutativity rule CR. Thus, s' is reducible, a contradiction. Consequently, our assumption was false, and $s \in$ PRED.

"$\subseteq$:" Conversely, let $s \in$ PRED, and assume that $s \notin$ LRC $\cap$ CSR. If $s \notin$ CSR, s cannot be in PRED by definition, so it suffices to assume that $s \in$ CSR–LRC. Suppose s contains a conflict of the form $w_i(x)w_j(x)$. Since $s \notin$ LRC, either

(a) t_j commits and either t_i does not commit or commits after t_j, or

(b) t_i aborts and either t_j does not abort or aborts after t_i.

Consider (a): If t_j commits and t_i does not commit, then we have $w_i(x) <_s w_j(x) <_s c_j$. Therefore, in exp$(s)$ we have $w_i(x) < w_j(x) < w_i^{-1}(x)$, which is not reducible, a contradiction. If both t_j and t_i commit and $c_j <_s c_i$, then the prefix of s containing c_j but not c_i is not reducible, by the same reason as in the previous case, which again yields a contradiction.

Next consider (b): Assume that t_i aborts and t_j does not abort. Since t_i and t_j are in active conflict, we have that $w_i(x) <_s w_j(x) <_s a_i$. If t_j commits, it follows that $w_i(x) < w_j(x) < w_i^{-1}(x)$ in exp(s), which is not reducible. If t_j does not terminate in s, we find $w_i(x) < w_j(x) < w_i^{-1}(x) < w_j^{-1}(x)$ in exp(s), which is also not reducible. So we are again faced with a contradiction. Finally, if both t_i and t_j abort s.t. $a_i <_s a_j$, then the prefix of s containing a_i but not a_j is not reducible by the same reason as in the previous case, which again yields a contradiction to s being prefix-reducible.

Similarly it can be shown that s must also be recoverable; in particular, note that a prefix of the form $w_i(x) <_s r_j(x) <_s c_j$ is not prefix-reducible. This concludes the proof of Theorem 11.30. □

The important point about Theorem 11.30 lies in the fact that there is now a syntactically testable condition available which together with CSR can characterize all of PRED. In particular, this condition places two rules on termination actions, which, as will be seen below, can easily be incorporated into scheduling protocols. Thus, the characterization of PRED provided by Theorem 11.30 is indeed a constructive one.

11.5 Further Relationships Among Correctness Criteria

We now collect a variety of further relationships between the classes of histories introduced in the previous subsections. As we have seen already, RG is the only class of histories among the "syntactic" ones which is comparable to CSR w.r.t. set inclusion. In other words, recoverability, avoiding cascading aborts, and strictness are properties of histories that are *orthogonal* to serializability; thus, a history may be, say, strict *without* being conflict-serializable. As a consequence, two criteria are required in order to make history and schedule "correctness" precise, as long as serializability and anything strictly "above" RG is preferable. This is no longer true for RG, clearly at the expense of disallowing some histories.

As will be seen from the first theorem we prove in this subsection, RG is, in a sense, in line with the class of prefix-reducible histories, as these also fall into the intersection of the classes of conflict-serializable and of recoverable histories, so that rigorousness or prefix-reducibility *alone* suffices as a correctness criterion for histories. However, class PRED is somewhat "better off" than RG, since the intersection of CSR and ST is *contained* in PRED, not vice versa. In detail, we can state the following theorem, a result that underlines the general importance of prefix reducibility in the context of database transactions:

Theorem 11.32

$$\text{CSR} \cap \text{ST} \subset \text{PRED} \subset \text{CSR} \cap \text{RC}$$

PRED vs. ST and RC

Proof. Let us look at the left inclusion first and suppose that history s is both in CSR and ST. Let s' be an arbitrary prefix of s; since CSR and ST are both prefix-closed, it follows that $s' \in$ CSR $\cap$ ST. Now assume that $s' \notin$ RED. Then exp(s') cannot be transformed into a serial history using rules CR, NR, UR, and OR alone. However, since s' itself *is* serializable, this can only be caused by do/undo pairs of the form $(w_i(x), w_i^{-1}(x))$ that cannot be brought together by the transformation rules. Thus, there must exist an action p_j, $p \in \{r, w\}$, $j \neq i$ which is in conflict with $w_i(x)$ and with $w_i^{-1}(x)$, and which must be ordered as follows:

$$w_i(x) < p_j < w_i^{-1}(x)$$

We now consider both possible cases individually:

(i) $w_i(x) < r_j(x) < w_i^{-1}(x)$: In this case, t_j must terminate with a commit operation, since the read operation would otherwise have been eliminated from the history by rule NR. However, this contradicts the fact that $s' \in$ ST, since t_j can read item x only after t_i is finished.

(ii) $w_i(x) < w_j(x) < w_i^{-1}(x)$: This immediately gives a contradiction to the fact that s' is strict.

As both cases lead to a contradiction, our assumption that $s' \notin$ RED was false, which concludes the first portion of the proof.

For the second inclusion stated by the theorem, consider a history $s \in$ PRED. If s was not in CSR, the conflict graph of s would contain a cycle of the form $t_1 \to t_2 \to \ldots \to t_n \to t_1$ involving a committed transaction. But then s cannot be transformed into a serial history by the reduction rules, which contradicts our assumption that $s \in$ PRED.

Next assume that $s \notin$ RC. This can only happen in one of the following situations:

(i) $w_i(x) < r_j(x) < c_j < c_i$: In this case, the prefix comprising the first three operations only is not reducible, a contradiction.

(ii) $w_i(x) < r_j(x) < c_j < a_i$: Here, the argument is the same as in case (i).

(iii) $w_i(x) < r_j(x) < a_i < c_j$: In this case, the expanded schedule cannot be prefix-reducible, again a contradiction to the assumption $s \in$ PRED. □

Figure 11.3 gives an even more complete picture of the landscape of correctness criteria that refer to expanded schedules. Clearly, the "uniformity" exhibited by prefix-reducibility is desirable, as a scheduler no longer needs to observe two distinct properties during operation; instead, it can now be sure that its output is correct as long as a single property is guaranteed.

Figure 11.3 also mentions the relationship between rigorousness and membership in XCSR, which is based on the following observation:

Theorem 11.33

RG $\subset$ PXCSR

RG vs. PXCSR

Proof. We already know that RG $\subset$ ST. We now show that a history $s \in$ RG has an acyclic conflict graph: Suppose, on the contrary, that for some $s \in$ RG the conflict graph $G(s)$ has a cycle, say, of the form $t_1 \to t_2 \to \ldots \to t_k \to t_1$. Since s is rigorous, there is always a commit or abort operation for transaction t_i between an operation p_i from t_i and a conflicting operation q_j from t_j, $j \neq i$. Extending this argument via induction to the cycle, it follows that there is a termination operation for t_1 between an operation from t_1 and a conflicting subsequent operation from the same t_1, a contradiction. As a consequence, $G(s)$ is acyclic.

We next show that a history from ST with an acyclic conflict graph is in PXCSR (which then implies the statement of the theorem). So suppose $s \in$ ST is such that $G(s)$ is acyclic. Consider an arbitrary prefix s' of s. As we have mentioned already, as ST is prefix-commit closed, $s' \in$ ST, and, as a subhistory of s, also has an acyclic conflict graph. It remains to be shown that $s' \in$ XCSR. If s contains committed transactions only, we have $s = \exp(s)$, and, by the definition of expanded conflict serializability, that $s \in$ XCSR. If s contains one or more aborted transactions, let us assume that $s \notin$ XCSR. In this case, $\exp(s)$ is not conflict-serializable, and the conflict graph of $\exp(s)$ contains at least one cycle; assume there is just one cycle. Since $G(s)$ is acyclic according to our general assumption, the cycle in $G(\exp(s))$ can only be caused by a newly added undo operation w_i^{-1} from an aborted transaction t_i. Now consider two arbitrary operations w_i and p_j in conflict, $i \neq j$; clearly, p_j is in conflict with w_i^{-1} as well. However, if $p_j < w_i < w_i^{-1}$ or $w_i < w_i^{-1} < p_j$, no new edge will be added to the conflict graph of $\exp(s)$, since a corresponding conflict edge between the two transactions involved is already present in the graph. So the only situation which can add an edge is $w_i < p_j < w_i^{-1}$, which is forbidden due to the fact that $s' \in$ ST (t_i must be finished prior to p_j).

Thus, we can conclude that $s \in$ XCSR, and, as s' was chosen arbitrarily, that $s \in$ PXCSR. $\square$

This theorem again confirms what was implied by Theorem 11.28 already, namely that RG $\subset$ CSR.

A simple consequence of the above, which can be shown along the lines of the proof of the previous theorem, is the following:

Theorem 11.34

PXCSR = ST $\cap$ XCSR

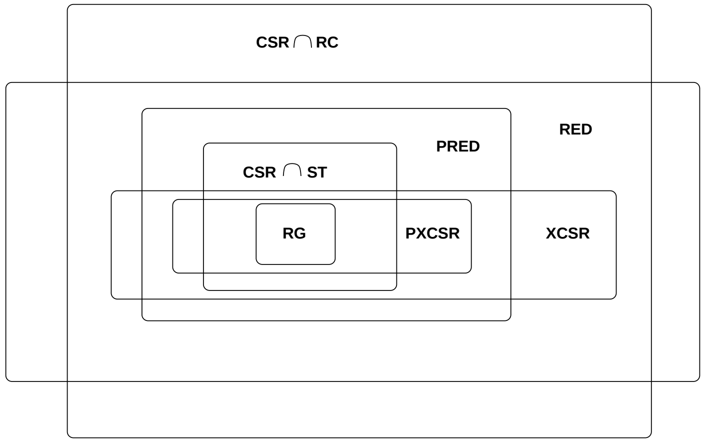

Figure 11.3: Relationship between reducible histories and strict ones.

11.6 Extending Page-Model Concurrency Control Protocols for Schedules with Transaction Aborts

In this section we present extensions of traditional concurrency control protocols we have studied in Chapter 4, suited to recognize histories in the classes we have introduced in this chapter. We mainly concentrate on three of them, ST, RG, and PRED, as these have turned out to be most desirable.

11.6.1 Extending Two-Phase Locking for Strictness and Rigorousness

Recall the 2PL protocol from Chapter 4, a simple, yet effective protocol for generating histories in CSR. In Section 4.3.4., we briefly mentioned several variants of the basic 2PL protocol, among them *strict* 2PL (S2PL), which holds write locks of a transaction until that transaction terminates, and *strong* 2PL (SS2PL), which holds *all* locks of a transaction until that transaction terminates. Moreover, we have seen already that

- *Gen*(SS2PL) $\subset$ *Gen*(S2PL) $\subset$ *Gen*(2PL), and
- *Gen*(SS2PL) $\subset$ COCSR.

Earlier in this chapter we additionally found out that

- RG $\subset$ COCSR

We next complete this picture with the following result:

SS2PL produces exactly RG

Theorem 11.35

Strong two-phase locking (SS2PL) generates exactly the class of rigorous schedules, i.e., *Gen*(SS2PL) = RG.

The proof is left as an exercise to the reader (see Exercise 7).

If we relax SS2PL to S2PL, we arrive at the following result (which can serve as a justification for the name "strict 2PL"):

Theorem 11.36

Strict two-phase locking (S2PL) generates only schedules that are both CSR and strict, i.e., *Gen*(S2PL) $\subseteq$ CSR $\cap$ ST.

Proof. Consider a history s produced by an S2PL scheduler. Then for $w_i(x) <_s o_j(x)$, $i \neq j$, $o \in \{r, w\}$ we first find

(i) $wl_i(x) < w_i(x) < wu_i(x)$, and

(ii) $ol_i(x) < o_j(x) < ou_i(x)$

due to the fact that an S2PL scheduler is in particular a 2PL scheduler. Since $wl_i(x)$ and $ol_i(x)$ collide, it follows that either (a) $wu_i(x) < ol_j(x)$, or (b) $ou_j(x) < wl_i(x)$. However, (b) together with (i) or (ii) leads to a contradiction ($o_j(x) < w_i(x)$). Hence (a) applies. Now if the scheduler keeps write locks until transaction termination, it follows that either $a_i < wu_i(x)$ or $c_i < wu_i(x)$, which means that either $a_i < o_j(x)$ or $c_i < o_j(x)$. The claim of the theorem follows. □

11.6.2 Extending Serialization-Graph Testing for Log Recoverability

We next look at concurrency control protocols, or extensions thereof, capable of generating histories in class PRED. The general basis of these extensions is the

characterization we have seen in Theorem 11.30 which states that PRED equals the intersection of CSR and LRC, the class of all log-recoverable histories. We study an extension of the SGT protocol first and then briefly look at other protocols for which the extensions are similar.

The basic idea is to add sufficient bookkeeping to recognize essential orderings of commit and abort events that need to be enforced. The enforcement itself is based on *deferring commits* and enforcing *cascading aborts* whenever necessary. In essence, a write-read or write-write conflict from transaction t_i to t_j creates a commit-ordering dependency, so that t_j must commit only after t_i. If t_j requests its commit before t_i is committed, t_j's commit request must wait. Conversely, if t_i aborts (in which case t_j must not yet be committed), it requires aborting t_j as well, and that abort must be performed first.

basic idea: deferred commits and cascading aborts

To develop the details of this approach, consider a history s. We use a refinement of the ordinary conflict graph $G(s)$ that (i) considers conflicts between any two transactions, not only committed ones, and in which (ii) the edges are annotated with the respective type of conflict (rw, wr, or ww); let us call this refinement the *fully annotated* conflict graph. What we are aiming at is a protocol that, according to the definition of LRC, produces histories in RC (i.e., watches wr conflicts) and in addition takes appropriate care of ww conflicts. So what we actually want to look at is a *partially annotated* conflict graph which is obtained from the fully annotated one by dropping all edges that do *not* correspond to wr or ww edges (and also dropping all nodes that do not have any incoming or outgoing edges). .

annotated conflict graph

We can now restate the LRC property as follows: A history s is in class LRC if for every edge of the form (t_i, t_j) in the partially annotated conflict graph the following holds: If t_i commits, then it commits before t_j does so, and if t_i aborts, then either t_i and t_j abort jointly in a group abort or t_i aborts after t_j aborts. As a history in PRED must be in LRC and CSR, we additionally need that, for a given history s, the ordinary conflict graph of the committed projection of s is acyclic.

LRC revisited

We next define two sets of transactions related to an arbitrary transaction taken from a partially annotated conflict graph $G(s) = (V, E)$:

(i) For $t_j \in V$, let precede$(t_j) := \{t_i \in V \mid (t_i, t_j) \in E\}$;
in other words, precede(t_j) contains the immediate predecessors of transaction t_j.

(ii) For $t_j \in V$, let follow$(t_j) := \{t_k \in V \mid t_k$ can be reached from $t_j\}$;
thus, follow(t_j) contains the transactions that are direct or transitive successors of t_j.

With these preparations, we are ready to outline a (pessimistic) serialization-graph tester as follows:

1. When an operation o_i that is neither an abort nor a commit operation arrives, the scheduler creates a node for t_i in the serialization graph, if it is not already there; in addition, the scheduler adds all relevant ww, wr, and rw edges. If a cycle occurs, transaction t_i is aborted, i.e., a_i is executed.
2. When a commit c_i arrives, the scheduler checks whether precede$(t_i) = \emptyset$. If this is the case, c_i is executed; afterwards all commit operations of all successors of t_i that are waiting to commit are scheduled for execution. Then t_i is removed from all immediate predecessor lists as well as from all successor lists. If t_i is a source in the serialization graph, it can also be removed from the graph.

   If precede$(t_i) \neq \emptyset$, t_i is placed in a queue of transactions waiting to commit.
3. When an abort a_i arrives, the scheduler checks whether follow$(t_i) = \emptyset$; if so, a_i can be executed. Otherwise, a_j is submitted for execution for every transaction $t_j \in$ follow(t_i); thereafter, all aborted transactions are removed from a queue of transactions waiting to commit, from the serialization graph, and from all immediate predecessor and successor lists.

Extended SGT

Let us call the above protocol *Extended SGT* or ESGT for short. Note that it is possible that an abort operation is issued when the transaction in question has already been aborted as a result of cascading aborts. It is also assumed in this protocol that the scheduler has some mechanism for ensuring that no two conflicting operations are processed at the same time.

The correctness of ESGT is established by the following theorem:

Theorem 11.37

Gen(ESGT) $\subseteq$ PRED

Proof. We use the characterization of membership in class PRED given in Theorem 11.30: By Rule 1, each history generated by ESGT is conflict-serializable. With respect to the LRC properties, we observe the following: In case of a direct conflict of the form

$$w_i(x)r_j(x),$$

$c_i < c_j$ is ensured by Rule 2. This is because t_j cannot commit until precede(t_j) is empty. By definition, $t_i \in$ precede(t_j), so t_i must have been committed or aborted (in the latter case, t_j will also be aborted). Thus, ESGT produces histories in RC.

In case of a direct conflict of the form

$$w_i(x)w_j(x),$$

we have that

(i) either $c_i < c_j$ (by Rule 2, similar to the previous case)

(ii) or $c_i < a_j$ (once t_i commits, it does not impose any further restrictions on t_j)

(iii) or $a_j < c_i$ (as Rule 3 does not depend on the precede set)

(iv) or $a_j < a_i$ (an abort of t_j does not affect t_i at all, so t_i can still be committed or aborted once t_j aborts).

No other combination is possible, since Rule 2 does not allow t_j to commit until t_i terminates, and Rule 3 forces the abort of t_j before t_i can be aborted. □

Example 11.38

Suppose that $w_1(x)w_2(x)w_2(y)$ has already been executed, and that $w_1(y)$ arrives at the scheduler. Clearly, the scheduler finds a cycle in the conflict graph, so $w_1(y)$ is rejected, and by Rule 1 a_1 is executed instead. Moreover, the scheduler finds follow(t_1) = $\{t_1, t_2\}$, so by Rule 3 a_2 is executed as well. □

Example 11.39

Suppose that $w_1(x)r_2(x)$ has been executed, and that c_2 is submitted. The scheduler now finds precede$(t_2) = \{t_1\}$. Therefore, t_2 is put on the queue of transactions waiting to be committed (Rule 2). After that, c_1 is submitted for execution. Since precede(t_1) = $\emptyset$, c_1 is executed right away. Thereafter, t_2 is removed from the waiting queue, and c_2 is submitted for execution. □

The above protocol is "pessimistic" in that a cycle test on the conflict graph is executed frequently, i.e., whenever a new edge has been added. An optimistic variant of the protocol would perform such a test less frequently, e.g., only upon processing a transaction's commit request; this can be accomplished at the expense that processing a commit gets slightly more complex.

11.6.3 Extending Other Protocols for Log Recoverability

A protocol along the lines of ESGT presented in the previous subsection can be implemented to recognize all LRC (and hence all PRED) histories. However, every SGT-style protocol incurs significant overhead related to the proper maintenance of the conflict graph. So an alternative would be to extend the two-phase locking (2PL) protocol for PRED, again on the basis of its characterization through LRC.

Extended 2PL (E2PL)

In order to guarantee that the basic 2PL protocol generates histories in class LRC, the protocol must be extended with additional rules that constrain the ordering of commits and aborts of all transactions. To this end, we again assume the maintenance of sets precede(t_i) and follow(t_i) for each transaction t_i as for the ESGT protocol. The extension of 2PL is then obtained by combining standard 2PL with Rules 2 and 3 of the ESGT protocol.

It can be shown that not every LRC history can be generated in this way. For example, history $w_1(x)w_2(x)w_3(y)w_1(y)c_3c_1c_2$ is in LRC, but not in *Gen*(E2PL), since t_1 would have to release the write lock on x before acquiring the write lock on y.

Extended TO (ETO)

Another protocol that can be considered for extension is the timestamp-ordering (TO) protocol. Once again, the idea is to combine the standard TO rule with Rules 2 and 3 from the ESGT protocol presented above, and again the result is such that some histories from LRC are missed. For example, $w_2(x)w_1(x)c_2c_1$ is in LRC, but not in *Gen*(ETO), if we assume that t_1 has a smaller timestamp than t_2.

11.7 Correctness Criteria for the Object Model

In this section, we extend our considerations regarding transaction recovery by looking at correctness criteria for the object model of transactions. We do this no longer in the generality we have studied correctness for the page model, but restrict the attention to practically relevant cases.

Essentially and as before, there are two types of object schedules to look at: flat ones as well as general ones. We will look at flat ones first and indicate that the theory developed earlier in this chapter vastly carries over to this model. In particular, the notions of expanded serializability and prefix reducibility can readily be adapted to this model. Thus, we can again establish rules (such as an "undo rule") for testing whether operations and their inverses can safely be eliminated from a given history or schedule, and we can design scheduling protocols based on appropriate characterizations. For general object schedules, it may be possible to apply such rules in a level-by-level fashion, but we can also refer back to the syntactic conditions we have earlier for page-level schedules.

11.7.1 Aborts in Flat Object Schedules

inverse operations (compensating steps)

To generalize expanded serializability as well as prefix-reducibility to flat object schedules, all we need to do is to make sure that undo rule UR can be applied to semantically richer operations as well. Undoing an operation such as withdraw simply amounts to invoking an *inverse operation*; such inverses have also been coined *compensating steps* in the literature. The inverse of an operation depends on the type of the original operation, its input parameters, and often also its output parameters. Thus, inverses are derived dynamically when the "forward" operation to which an inverse refers executes, as opposed to a static definition of operation pairs. For example, the inverse of a successful withdrawal consists of depositing the same amount of money back into the account, and the inverse of an unsuccessful withdrawal (i.e., a withdraw operation with return value "no") is empty and can be viewed as a special "null operation" without any effects. It is important to note that inverses refer to the same abstract view that underlies the commutativity considerations of the corresponding forward operations. For example, if an unsuccessful withdrawal leaves an entry in an audit trail, depositing the money back could be viewed as an appropriate inverse even if it does not remove that entry. It is for this reason that some people prefer the term "compensating step" as we do not necessarily need "exact inverses" that undo each and every detail of the forward operations' effects.

inverse operation

Definition 11.40 (Inverse operation)

An operation $f'(x'_1, \ldots, x'_{m'}, \uparrow y'_1, \ldots, \uparrow y'_{k'})$ with input parameters x'_1 through $x'_{m'}$ and output parameters y'_1 through $y'_{k'}$ is the *inverse operation* of the operation $f(x_1, \ldots, x_m, \uparrow y_1, \ldots, \uparrow y_k)$ if for all possible sequences α and ω of operations on a given interface (e.g., an object type), the return parameters in the concatenated sequence

$$\alpha f(\ldots) f'(\ldots) \omega$$

are identical to those in the sequence

$$\alpha\omega.$$

We will also write $f^{-1}(\ldots)$ rather than $f'(\ldots)$ and will omit parameters whenever the context is clear.

prefix reducibility

Once a method for deriving the inverse operation from the input and output parameters of an executed forward operation is defined, all our prior considerations on expanded schedules and prefix-reducibility carry over in a straightforward way.

Example 11.41

Consider the following schedule of two funds-transfer transactions:

$$s = withdraw_1(a)\ withdraw_2(b)\ deposit_2(c)\ deposit_1(c)\ c_1\ a_2.$$

The expanded schedule that makes the undo steps for the abort of transaction t_2 explicit looks as follows:

$$exp(s) = withdraw_1(a)\ withdraw_2(b)\ deposit_2(c)\ deposit_1(c)\ c_1\ reclaim_2(c)\ deposit_2(b)\ c_2,$$

where the *reclaim* operation is an unconditional withdrawal without overdraft protection, thus commuting with deposits. This expanded schedule can be reduced to a serial order of t_1 and the empty transaction t_2, or equivalently to t_1 alone. As a consequence, s is reducible. □

The complete transaction trees for the expansion of Example 11.41, including the underlying page-level actions for the isolated object-level operations, are depicted in Figure 11.4. Note that the high-level inverse operations do not have page-level children in the expansion. In the actual implementation, the inverse operations will, of course, again spawn lower-level page operations, but these are created dynamically during their execution and not precisely known in advance. For example, the $reclaim_2(c)$ operation issued on behalf of t_2's abort may actually access a page q different from the page p that its corresponding forward operation accessed. This may occur if the record that represents the account object c has been moved to a different page by a low-level storage reorganization in between the forward and the inverse operation (see Chapter 1 for a brief outline of such storage management issues).

Figure 11.5 illustrates why, in Example 11.41, the abort of t_2 cannot be simply implemented by issuing inverse page-level writes to undo the t_2's effects. The resulting, incorrectly "expanded" schedule is not tree-reducible, and there is no way of commuting the page-level writes such that the inverse writes of t_2 and its forward write steps become adjacent operations.

page-level undo is incorrect for object-model transactions

As the previous example indicates, object-level operations and their inverses or compensations need to be considered at the same level of abstraction, and compensation cannot simply be achieved by looking at the "technical" level of page writes.

Example 11.42

For another example, consider the following history:

$$s = insert_1(x) delete_2(x) insert_3(y) a_1 a_2 a_3.$$

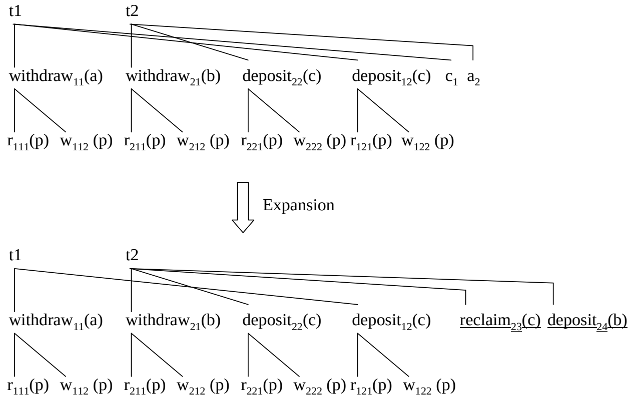

Figure 11.4: Example of an expanded schedule for the flat object model.

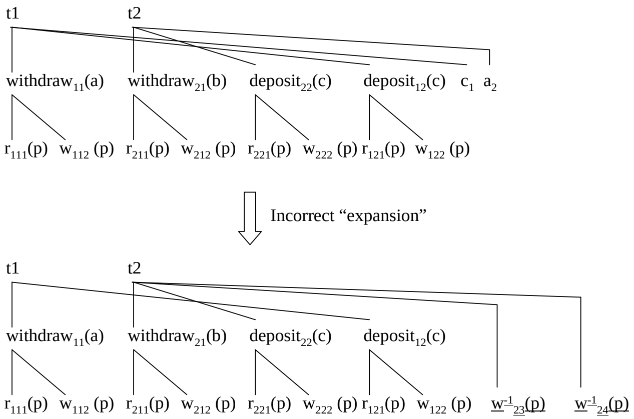

Figure 11.5: Incorrect "expansion" of the example of Figure 11.4.

We can expand s as before to obtain the following history:

$$\begin{aligned} exp(s) \quad = \quad & insert_1(x) delete_2(x) insert_3(y) insert_1^{-1}(x) c_1 \\ & delete_2^{-1}(x) c_2 insert_3^{-1}(y) c_3. \end{aligned}$$

Since $delete_2(x)$ and $insert_1^{-1}(x)$ do not commute, we cannot eliminate t_1 from $exp(s)$, so s is not reducible. Not that s is also not expanded serializable, as there is a conflict from t_1 to t_2 on x as well as one from t_2 to t_1, also on x. □

Thus, we may find flat object histories reducible or non-reducible according to the same rules as before, and the same applies to the notion of conflict serializability when that is based on commutativity or non-commutativity of operations. Thus, scheduling protocols can again be designed as generalizations of the protocols we have previously introduced, such as 2PL and its variants; especially the strict 2PL (S2PL) protocol turns out to be relevant in this context.

Locking protocols generally base their notion of conflicting locks on the corresponding notion of conflicting operations. For example, in the page model with its read and write operations, we have seen earlier that the ordinary commutativity relation is as follows, where "+" indicates commutativity, and "−" indicates the contrary:

	$r_j(x)$	$w_j(x)$
$r_i(x)$	+	−
$w_i(x)$	−	−

We can extend this table for inverse operations in a straightforward way, where it is assumed that an inverse read is actually an all-commutative "null operation" and can thus be omitted.

	$r_j(x)$	$w_j(x)$	$w_j^{-1}(x)$
$r_i(x)$	+	−	−
$w_i(x)$	−	−	−
$w_i^{-1}(x)$	−	−	−

The latter commutativity relation has an interesting property: If two operations p and q commute, then all other combinations of these operations or their inverses, i.e., p with q^{-1}, p^{-1} with q, and p^{-1} with q^{-1}, commute as well, provided that none of these operations is a null operation (which trivially commutes with every other operation). Moreover, if p and q do not commute, the same holds for all their other combinations. A commutativity relation exhibiting this property is called *perfect*. In the general setting of object-level schedules, we formally have:

perfect commutativity

Definition 11.43 (Perfect commutativity)

Given a set of operations for an object type, such that for each operation $f(x, p_1, \ldots, p_m)$ an appropriate inverse operation $f^{-1}(x, p'_1, \ldots, p'_{m'})$ is included. A commutativity table for these operations is called *perfect* if the following holds:

$f(x, p_1, \ldots, p_m)$ and $g(x, q_1, \ldots, q_n)$ commute
$\Leftrightarrow$ $f(x, p_1, \ldots, p_m)$ and $g^{-1}(x, q'_1, \ldots, q'_{n'})$ commute
$\Leftrightarrow$ $f^{-1}(x, p'_1, \ldots, p'_{m'})$ and $g(x, q_1, \ldots, q_n)$ commute
$\Leftrightarrow$ $f^{-1}(x, p'_1, \ldots, p'_{m'})$ and $g^{-1}(x, q'_1, \ldots, q'_{n'})$ commute

For object types with perfect commutativity, S2PL is directly applicable. A particularly nice property of perfect commutativity becomes obvious in the context of deadlock resolution. When we choose a transaction as a deadlock victim and initiate its rollback by means of appropriate inverse operations, these inverse operations could, in principle, request additional locks. Consequently, a transaction that is being rolled back could again become involved in a deadlock. This is not a situation that could not be handled at all, but to ensure freedom of starvation for deadlock victims it is most desirable to avoid acquiring additional locks during rollback. With perfect commutativity, such additional locks are unnecessary, for the locks that have been acquired on behalf of the transaction's regular "forward" operations are sufficient for the inverse operations as well. Of course, this consideration does not hold for lower-level locks that are needed within a subtransaction (e.g., short-term page locks for the duration of a high-level inverse operation). Such short-term locks will still have to be acquired during rollback, but usually do not present a problem regarding starvation.

deadlock-freedom during rollback

We next show that for semantically rich operations, perfect commutativity may not always be achievable:

Example 11.44

Consider the following operations on an object of type "set":

1. $insert(x)$ inserts element x into a given set and returns 1 if x was actually inserted, and 0 otherwise (x was already contained in the set),
2. $delete(x)$ deletes element x from a given set, and returns 1 if x was actually deleted, and 0 otherwise (x was not present in the set),
3. $test(x)$ returns 1 if x is a member of a given set, and 0 otherwise.

Let inverses for these "forward" operations be defined as follows:

1. $insert^{-1}(x)$, where x is either the value that was inserted by the corresponding forward operation or 0 if the value was present already, deletes x from the given set if $x \neq 0$, and does nothing otherwise; it always returns 0.
2. $delete^{-1}(x)$, where x is either the value that was deleted by the corresponding forward operation or 0 if the value was not present, inserts x if $x \neq 0$, and does nothing otherwise; it also returns 0.
3. $test^{-1}(x)$ is a null operation that will not be further considered.

The commutativity relation for these operations is shown in the following table:

	$insert(x)$	$delete(x)$	$test(x)$	$insert^{-1}(x)$	$delete^{-1}(x)$
$insert(x)$	–	–	–	–	–
$delete(x)$	–	–	–	–	–
$test(x)$	–	–	+	–	–
$insert(x)^{-1}$	–	–	–	+	–
$delete^{-1}(x)$	–	–	–	–	+

Notice that, for example, $insert(x)$ does not commute with itself do to the different return values resulting from an attempt to insert the same element twice. On the other hand, $insert^{-1}(x)$ *does* commute with itself, and similar comments apply to *delete*. Thus, the commutativity table shown is not perfect. □

From a lock-based scheduler's point of view, for non-perfect commutativity an inverse operation may need an additional, stronger lock that has not already been obtained with the lock for its corresponding forward operation. This problem can be rectified by forcing the lock manager to preclaim the necessary stronger lock already with the forward operation, effectively using a somewhat coarser version of the lock compatibility table derived as follows:

perfect closure of a lock compatibility table

Definition 11.45 (Perfect closure)

The *perfect closure* of a commutativity table for the operations of a given object type is the largest, perfect subset of the original commutativity table's commutative operation pairs.

Analogously, the perfect closure of the corresponding lock compatibility table contains the smallest number of "–" entries such that all "–" entries of the original lock compatibility table are included and for each operation pair $f(\ldots)$ and $g(\ldots)$, either all four combinations among $f(\ldots)$, $f^{-1}(\ldots)$, $g(\ldots)$, and $g^{-1}(\ldots)$ have "+" entries or all four have "–" entries.

Thus, considering the perfect closure of a commutativity table that is not perfect is a convenient "workaround" for a scheduler. It does not need additional locks for inverse operations, as these are already covered by the locks acquired for the corresponding forward operations, and the potential loss in concurrency compared to using the original, non-perfect, commutativity table is marginal in practice. Also, as mentioned earlier, this way there is no danger that rolling back a transaction as a deadlock victim can again lead to deadlocks.

For the operations of Example 11.44, we obtain the following perfect closure:

	$insert(x)$	$delete(x)$	$test(x)$	$insert^{-1}(x)$	$delete^{-1}(x)$
$insert(x)$	−	−	−	−	−
$delete(x)$	−	−	−	−	−
$test(x)$	−	−	+	−	−
$insert(x)^{-1}$	−	−	−	−	−
$delete^{-1}(x)$	−	−	−	−	−

Although it may appear that, by having replaced two + entries with −− entries, resorting to the perfect closure could result in significantly more lock conflicts, we should note that this is rarely the case as only inverse operations are affected and transaction aborts should be infrequent anyway.

Another restriction on commutativity relations that is closely related to the notion of perfectness is the following:

normal commutativity

Definition 11.46 (Normal commutativity table)

A commutativity table for a given object type is *normal* if for all operations $f(\ldots)$ and $g(\ldots)$ the following holds: if $f(\ldots)$ does not commute with $g(\ldots)$, then $f^{-1}(\ldots)$ does not commute with $g(\ldots)$ and $f(\ldots)$ does not commute with $g^{-1}(\ldots)$ (provided that these inverse operations are not null operations, in which case commutativity is self-guaranteed).

For example, the commutativity relation from Example 11.44 is not normal, which is easily verified just by looking at $f = g = insert$ and the table shown earlier. Obviously, every perfect commutativity table is also normal, but the converse does not hold. The property is of theoretical interest because it can be shown (see the literature cited in the bibliographical remarks) that the classes PRED and CSR∩LRC coincide for normal commutativity tables, which generalizes Theorem 11.30 to the flat object model. The same holds for perfect commutativity, commutativity is a weaker prerequisite for the equality of the two classes PRED and CSR∩LRC.

At this point, we have all the ingredients to apply prefix reducibility to flat object schedules, but we know well that this is not a constructive method to build practical schedulers. So, in perfect analogy to our considerations in the page model,

we should now look out for syntactic, easily implementable, conditions that imply prefix reducibility. In doing this, we are willing to sacrifice generality a little bit by assuming that commutativity tables are perfect. From the above mentioned generalization of Theorem 11.30 we know that conflict serializability in combination with log recoverability would be such a syntactic condition, and that this one would even be sufficient and necessary (under the assumptions about commutativity tables). However, notwithstanding our sketch of LRC-based extended concurrency control algorithms for the page model in Section 11.6, practical systems often require simpler methods with very low overhead. For this reason, the most intriguing syntactic criterion towards such practically viable protocols is the notion of strictness. In the flat object model, strictness needs to be redefined as follows:

strictness

Definition 11.47 (Strictness)

A flat object model schedule s with (perfect commutativity table) is *strict* if for all L1 actions $p_i(\ldots)$ and $q_j(\ldots)$ of transactions t_i and t_j, $i \neq j$, such that p_i is an update operation (i.e., not a read-only operation) and $p_i(\ldots)$ and $q_j(\ldots)$ do not commute, the ordering $p_i <_s q_j$ implies that either $c_i <_s q_j$ or $a_i <_s q_j$.
As in the page model, the class of strict schedules is denoted as ST.

Theorem 11.48

A flat object schedule s (with perfect commutativity table) that is conflict serializable and strict is also prefix reducible.

11.7.2 Complete and Partial Aborts in General Object-Model Schedules

In the general object model, a complication is that not only entire transactions may abort and have their inverse operations interleaved with other transactions, but also subtransactions may abort without rolling back the entire transaction. This option, sometimes called a *partial rollback* (as far as the transaction root is concerned), is available for every non-leaf operation in a transaction tree. From a performance viewpoint, partial rollbacks are an attractive option as they can be used, for example, to resolve a deadlock between two subtransactions and later automatically restart the deadlock victim without resorting to an abort of the entire transaction. As far as our formal model and correctness reasoning is concerned, the first preparation to cope with this additional complication is to attach a termination operation, commit or abort, to each subtransaction as follows:

Definition 11.49 (Terminated subtransactions)

An object-model history has *terminated subtransactions* if each non-leaf node p_ω has either a child $a_{\omega\nu}$ or a child $c_{\omega\nu}$ that follows all other children of p_ω (assuming that there are $\nu - 1$ such children). So each inner-node operation has a termination operation, abort or commit, for the corresponding subtransaction. An object-model schedule has terminated subtransactions if it is a prefix of an object-model history with terminated subtransactions (see Chapter 6 for the proper notion of prefix in the context of transaction forests).

In the following we consider only histories and schedules that have terminated subtransactions.

expanded schedule with partial aborts

Definition 11.50 (Expanded object-model schedule)

Let s be an object-model schedule with terminated subtransactions. The *expansion* of s, denoted $\exp(s)$, is an object-model schedule that is derived from s in the following way:

1. All operations whose parent has a commit child are included in $\exp(s)$.
2. For each operation whose parent p_ω has an abort child $a_{\omega\nu}$, an inverse operation for each of p's $\nu - 1$ forward operations is added provided the forward operations do themselves have a commit child, the abort child of p is removed, and a child $c_{\omega(2\nu-1)}$ is added to p. The inverse operations have the reverse order of the corresponding forward operations and follow all forward operations; the commit operation follows all other children of p. All new children of p precede an operation q in $\exp(s)$ if the abort child of p preceded q in s.
3. For each transaction in active(s), inverse operations and a final commit child are added as children of the corresponding transaction root, with their ordering defined analogously to the above case of an aborted subtransaction.

With this generalized kind of expansion, the resulting schedules look as if they were normal object-model schedules. Now the natural correctness criterion to be applied simply needs to adopt both notions of tree-reducibility and prefix-reducibility simultaneously by allowing all the underlying transformation rules as in the following definition:

tree-prefix-reducibility

Definition 11.51 (Tree-prefix-reducibility)

A object-model schedule s is called *tree-prefix-reducible (TPRED)* if its expansion, $\exp(s)$, can be transformed into a serial order of s's committed transaction roots by applying the following transformation rules finitely many times:

1. the commutativity rule applied to adjacent leaves,
2. the tree-pruning rule for isolated subtrees,
3. the undo rule applied to adjacent leaves,
4. the null rule for read-only operations, and
5. the ordering rule applied to unordered leaves.

As an example for an object-model schedule in which both a subtransaction and an entire transaction are aborted is shown in Figure 11.6.

The expanded schedule that results from the schedule of Figure 11.6 is given in Figure 11.7. The figure has inverse operations underlined to highlight them. As before, *reclaim* is an unconditional withdraw operation to compensate the effect of a deposit. For simplicity, we have omitted all commit steps of subtransactions and transactions in the figure, as the expansion has only commits and no aborts anyway. It is rather straightforward to reduce the expanded schedule to a serial order of t_1 and t_2. Removing empty subtransactions and transactions further reduces the schedule to the committed transaction t_1 alone.

Theorem 11.52

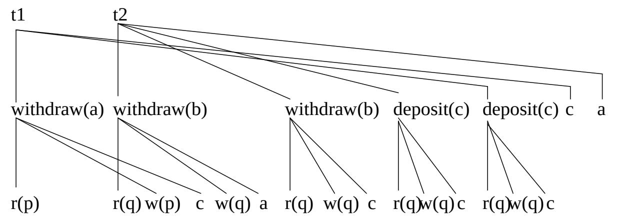

Figure 11.6: Example of an object-model schedule with both partial and complete aborts.

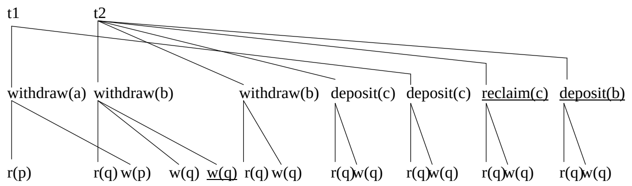

Figure 11.7: Expanded schedule for the example of Figure 11.6.

A layered object model schedule s (with perfect commutativity tables at all levels) is tree prefix reducible if all its level-to-level schedules are order-preserving conflict serializable and strict.

Theorem 11.53

A layered object model schedule s (with perfect commutativity tables at all levels) is tree prefix reducible if all its level-to-level schedules are conflict serializable and strict and s is conflict-faithful.

11.8 Extending Object-Model Concurrency Control Protocols for Schedules with Complete and Partial Transaction Aborts

Theorem 11.54

The layered S2PL protocol generates only schedules that are tree-prefix-reducible.

11.9 Lessons Learned

In this chapter, we have introduced and studied the theoretical foundations of transaction recovery. As we have seen, there are essentially two approaches: a "syntactic" one which merely pays attention to a proper sequencing of commit or abort operations, and a more "semantic" one which tries to make the actions that have to be performed in response to an abort operation as explicit as possible. The latter leads to interesting classes of histories, for example due to the fact that some fall into the intersection of the classes of conflict-serializable and of recoverable schedules, while CSR and RC are otherwise incomparable w.r.t. set inclusion. Moreover, the class PRED of prefix-reducible schedules, which is the most important one of the "semantic" categories, can be characterized in a way that gives rise to fairly straightforward extensions of standard concurrency control protocols. Thus, we have seen in this chapter that it is indeed possible to obtain an appropriate conceptual framework for transaction recovery that can even be easily built into known protocols.

In addition, the framework we have set up here for transactions in the page model can be carried over to the object model of transactions. To this end, the major issue is to properly define the "undo" operations for the object-level operations considered, and to determine what properties the corresponding commutativity relations have. As a result, this approach to transaction recovery is also suitable for layered transaction systems.

Exercises

1. Let $s_1 = w_1(x)w_1(y)r_2(u)w_2(x)r_2(y)w_2(y)a_2w_1(z)c_1$ and $s_2 = w_1(x)w_1(y)r_2(u)w_2(x)r_2(y)w_2(y)w_1(z)a_1c_2$.

   Determine $\exp(s_1)$ and $\exp(s_2)$ as well as the corresponding reductions.

2. Which of the properties RC, ST, RG, PRED, and LRC are satisfied by the following schedules:
   $s_1 = r_1(a)r_2(a)w_1(a)c_1c_2$
   $s_2 = r_1(a)w_1(a)r_2(b)w_2(b)w_2(a)c_2c_1$
   $s_3 = r_1(a)incr_2(a)incr_2(b)incr_3(b)c_3a_2c_1$

   Some of the above properties cannot be applied to a schedule with operations other than read and write, such as schedule s_3 above. Try to define generalizations of these properties to flat object schedules with arbitrary operations.

3. For each of the following histories determine to which of the classes RC, ACA, or ST it belongs:
   $s_1 = w_1(x)r_2(y)r_1(x)c_1r_2(x)w_2(y)c_2$
   $s_2 = w_1(x)r_2(y)r_1(x)r_2(x)c_1w_2(y)c_2$
   $s_3 = w_1(x)r_2(y)r_2(x)r_1(x)c_2w_1(y)c_1$

4. Show that membership in each of the classes RC, ACA, ST, RG, and PRED is a prefix-commit closed property.

5. Consider COCSR, the class of all histories that are commit-order-preserving conflict-serializable, and show the following:
   (a) COCSR $\cap$ RC $\neq \emptyset$, COCSR $\not\subseteq$ RC, RC $\not\subseteq$ COCSR
   (Thus, COCSR $\cap$ RC $\subset$ COCSR und COCSR $\cap$ RC $\subset$ RC.)
   (b) COCSR $\cap$ ACA $\subset$ ACA
   (c) COCSR $\cap$ ACA $\subset$ COCSR $\cap$ RC
   (d) COCSR $\cap$ ST $\subset$ ST
   (e) COCSR $\cap$ ST $\subset$ COCSR $\cap$ ACA

6. For each class intersection and each class difference in Figure 11.3, find a sample history.

7. Prove Theorem 11.35 stating that Gen(SS2PL) = RG.

8. Consider a database consisting of positive integers with the following operations defined on them:

   - $incr(x)$: increments x if $x > 0$ and returns 1; otherwise does nothing and returns 0.
   - $incr^{-1}(x, y)$: decrements x if y is the return value of the corresponding forward operation and $y \neq 0$; otherwise does nothing, always return 0.
   - $rest(x)$: reset x to 1 and returns the old value of x.
   - $reset^{-1}(x)$ set x to value y, where y is the return value of the corresponding forward operation; always returns 0.
   - $retrieve(x)$: returns the current value of x.
   - $retrieve^{-1}(x)$: is a null operation and returns an empty sequence.
   - $decr(x)$: decrements x and return 0.
   - $decr^{-1}(x)$: increments x and returns 0.

   Determine the commutativity relation for these operations, and find out whether it is perfect or normal. If one of the latter properties does not hold, try to restrict the relation in such a way that this respective property is achieved.

9. Prove Theorem 11.30 for a object model in which the relevant operations have a normal commutativity relation.

10. Consider an implementation of transaction recovery that makes use of transaction-private workspaces in that all updates are initially executed on such workspaces and thus deferred until the transaction's commit as far as the shared, persistent data is concerned. Upon the commit of a transaction, the workspace updates would be re-executed against the shared data; upon a transaction abort, the workspace would be discarded.

    Is it possible to apply the notions of ST, PRED, and LRC to such a workspace-based deferred-update implementation? Is it meaningful? If appropriate, try to redefine these properties for a deferred-update setting.

Bibliographical and Other Remarks

Historically, the purely syntactic criteria known as RC, ACA, and ST preceded the more general model based on expanded schedules and an explicit undo rule as part of a transformation-based correctness criterion. The formalization of the syntactic criteria goes back to Hadzilacos (1988), where they are discussed in detail. Our presentation of these criteria is based on Bernstein et al. (1987). Algorithmic and performance implications of these classes have been studied by Graham et al. (1984). The class of rigorous histories was introduced by Breitbart et al. (1991). A solution to Exercise 5 can be found in Raz (1992).

The concept of an expanded schedule that has led to a "unifying approach" to serializability and transaction recovery, on the other hand, has been introduced by Schek et al. (1993), building on earlier ideas from Moss et al. (1986) and Weikum (1986). Schek et al. (1993) also introduce the important notions of reducibility and prefix-reducibility. The syntactically oriented and thus more "constructive" criterion of log-recoverability has been presented by Alonso et al. (1994); in conjunction with conflict-serializability it has originally been named "serializability with ordered termination" and shown to be equivalent to prefix-reducibility in the page model and flat object model with perfect commutativity. Alonso et al. (1994) also discuss the extensions that are needed for the various classes of concurrency control algorithms in order to integrate transaction recovery.

Schek et al. (1993) considered a flat object model; this approach was later generalized by Vingralek et al. (1998) to non-perfect commutativity relations and by Hasse (1996) to the general object model. Solutions to Exercises 8 and 9 can be found in the former reference. Other work along the lines of the unifying model includes Moss et al. (1986), who discussed a subset of prefix-reducible schedules under the name "revocability", Weihl (1993), who has studied both a log-based model for transaction recovery and a workspace-based model where updates to persistent data are deferred until the commit point of a transaction, Rastogi et al. (1993), who has introduced the notion of strictness for schedules with semantically rich operations, and Levy et al. (1991) as well as Krychniak et al. (1996).

Papadimitriou (1986) as well as Papadimitriou and Yannakakis (1987) describe an approach to transaction recovery that is based on the idea that, even after a transaction has been aborted, a "reliable" schedule can be continued in such a way that ultimately a (view-) serializable history results; it has been shown, however, that the problem of deciding whether a given schedule is "reliable" is PSPACE-complete and hence of little practical relevance. The idea of leaving the semantics of an operation unchanged in every continuation of a given schedule is also taken up again by Lechtenbörger and Vossen (2000), who show that with minor adaptations of the traditional Herbrand semantics, classes of correct histories that strictly *contain* PRED can be established.

Soisalon-Soininen and Ylönen (1995) study a way to relax the RC and ACA requirements for histories through a slight variation of the transaction model. The idea is to allow reading and writing from a transaction that has not yet committed, but that has performed all its operations *and* has issued a "request" to commit. Under normal circumstances, such a transaction will thereafter commit, and the "partial strictness" obtained in this way is then roughly sufficient for membership

in RC or ACA. They add a new operation c^r, called *commit request*, to the set of possible operations of a transaction. In a transaction t_i, c_i^r may only occur as the second but last operation, i.e., the only operations that can still follow from t_i are a commit c_i or an abort a_i; conversely, if a transaction commits, i.e., ends with operation c_i, this c_i must be preceded by c_i^r. Next, a history s is *partially strict* if the following conditions hold for all transactions $t_i, t_j \in \text{trans}(s)$, $i \neq j$:

(i) if $w_j(x) <_s r_i(x)$, then $c_j^r <_s r_i(x)$,

(ii) if $w_j(x) <_s w_i(x)$, then $a_j <_s w_i(x)$ or $c_j^r <_s w_i(x)$,

(iii) if $c_j^r <_s c_i^r$ and $c_j \in s$, then $c_j <_s c_i$.

It can then be shown that if s is partially strict and for no transaction $t_i \in \text{trans}(s)$ we have $c_i^r <_s a_i$, then $s \in \text{RC}$.