Chapter 3

Concurrency Control: Notions of Correctness for the Page Model

"Nothing is as practical as a good theory." (Albert Einstein)

3.1 Goal and Overview of the Chapter

In this chapter we concentrate on the page model of transactions and develop notions of correctness for their concurrent execution. This study will reveal a variety of aspects: techniques for proving concurrency control algorithms correct, syntactic vs. semantic notions of correctness, NP-completeness results, and hierarchies of classes of correct histories, to mention just a few. Several of the notions presented in this chapter will later be re-used in the context of other models; more importantly, the *approach* of developing correctness criteria, which is based on establishing a relationship between concurrent and *serial* executions, has very few competitors. Indeed, although essentially invented more than 20 years ago, it is even relevant for studies done in recent years, and is often the starting point for novel developments and revisited viewpoints.

overview of the chapter

We will first model an execution of multiple transactions that is interleaved over time and thus concurrent by the notions of *schedules* and *histories*. For these notions two points will be important: First, executing transactions is a highly dynamic situation in reality; therefore, modeling this adequately might not be as easy as it may seem. In fact, the concurrency control component of a data server needs to be ready to decide on the execution of a newly arriving transaction in an instant, and

needs to synchronize it correctly with already running transactions. The technical tool will be to consider *prefixes* of histories, now called schedules. Second, we need to take into account that some transactions might not terminate successfully, but can be aborted (for reasons to be discussed later). For this purpose, we will follow common practice and introduce *termination operations* in addition to data operations that will just serve to terminate a transaction in one of two possible ways: either commit or abort. However, the formal treatment especially of aborts can easily become counterintuitive (see the bibliographical remarks for this chapter), which is why our formal exposition will mostly assume a failure-free environment.

We will take these techniques through a variety of notions of correctness, so the reader can become familiar with them and study how to apply them in distinct settings. Before doing so, we motivate the correctness discussion by some prominent synchronization problems.

3.2 Canonical Concurrency Problems

Before we embark on a formal discussion of correctness in the page model of transactions, let us show by way of examples that the model captures various of the most important patterns of concurrency problems. To this end, we consider three archetypical "conflicts" that may arise between the read and write operations of distinct transactions in the absence of proper concurrency control measures.

Example 3.1

The following problem is known as the *lost update problem*. Consider two transactions t_1 and t_2 which are concurrently executed as follows:

t_1	Time	t_2
	/* $x = 100$ */	
$r(x)$	1	
	2	$r(x)$
/* update $x := x + 30$ */	3	
	4	/* update $x := x + 20$ */
$w(x)$	5	
	/* $x = 130$ */	
	6	$w(x)$
	/* $x = 120$ */	
↑ update "lost"		

lost-update problem

We suppose that x is a numerical data item having a value of 100 at time 1. Both t_1 and t_2 read this value into local variables, modify the value in their local variables, and then each of the two writes back its local value into the shared data item. For simpler notation, we do not show the local variables of the processes explicitly, and rather refer to them by the name x in their local assignments. Assume that t_1 adds 20, while t_2 adds 30. So in the end x should have a value of 150. However, since t_2 is the last one to overwrite the original value of x, the final value is 120, which is incorrect. Indeed, because t_1 writes its new value back into the database before t_2 does, the former update is lost. □

Notice that it is irrelevant in the previous example what the two processes do with data item x once it has been transferred into their local variables; the important point is the uncontrolled overwriting of the first transaction's result by the second after the concurrent modifications of the local variables. For exactly this reason, we could afford a somewhat loose notation, without explicit names for the local variables. The essence of the problem really is the pattern of interleaved read and write operations:

$$r_1(x)r_2(x)w_1(x)w_2(x),$$

with time proceeding from left to right.

Such problems do not necessarily have to involve two "writers", as the following example with one read-only transaction shows.

Example 3.2

This problem is known as the *inconsistent-read problem*: Consider two data items x and y of type integer, with current values $x = y = 50$; so $x + y = 100$. This could arise in a banking application, where the data items represent account balances. For transfers among the accounts, their sum should obviously remain constant. Transaction t_1 below computes the current sum, while transaction t_2 transfers a value of 10 from x to y as follows:

inconsistent-read problem

t_1	Time	t_2
	1	$r(x)$
	2	/* $x := x - 10$ */
	3	$w(x)$
/* sum := 0 */	4	
$r(x)$	5	
$r(y)$	6	
/* sum := sum + x */	7	
/* sum := sum + y */	8	
	9	$r(y)$
	10	/* $y := y + 10$ */
	11	$w(y)$

Clearly, t_1 will return 90 as a result, which is wrong; however, this error cannot be easily recognized by the user who may thus be misguided in some follow-up real-world

actions. Again, the essence of the problem is the pattern of interleaved read and write operations:

$$r_2(x)w_2(x)r_1(x)r_1(y)r_2(y)w_2(y)$$

□

These two simple examples already give rise to two important observations:

1. In both cases, the effects are unwanted, although in principle not visible to the outside world;
2. a *serial* execution of the two transactions, i.e., an execution in which one transaction is executed completely before the other (either t_1 before t_2, or vice versa), would have avoided the erroneous effects; this observation will be used later to design a first correctness criterion.

Next we take a look at a problem where one of the transactions fails, and the transaction-specific rollback option is exerted (e.g., by the data server or by the transaction itself issuing an abort request), thus "erasing" all effects that the process has left on the permanently stored data.

Example 3.3

This example is known as the *dirty-read* problem or the problem of *reading uncommitted data*. Its major point is that one transaction fails after having modified some data, which is read by a second transaction in between the modification and the failure.

t_1	Time	t_2
$r(x)$	1	
/* $x := x + 100$ */	2	
$w(x)$	3	
	4	$r(x)$
	5	/* $x := x - 100$ */
`failure & rollback`	6	
	7	$w(x)$

dirty-read problem

So when transaction t_1 fails and is rolled back after the modified value of x has been read by t_2, this puts t_2 in an inconvenient position. Because of the failure, the modified value of x is not a committed one upon which t_2 can really rely. Nevertheless, this value is the only one that t_2 has read and will thus use for its own further processing, especially for computing the value that it will subsequently write back into x. For example, if x was a bank account, with an initial balance of 0 and no option for over-drafting the account (i.e., producing a negative balance), and t_1 was to deposit \$ 100 say in the form of a check, then t_2 would critically rely on the validity of this check and the deposit in its attempt to withdraw \$ 100 from this no-overdraft account. Obviously, the situation shown, where the withdrawal may eventually succeed but the deposit of the check fails, would severely damage the dependability and reputation of the bank's business. The solution would be that the value read by t_2 is rendered invalid by the rollback of t_1, and appropriate steps should be taken to prevent t_2 from depending on this invalid value. □

What can be seen from the above examples is that the data accesses performed by concurrently executing programs have a potential of "conflicting" with each other. Therefore, some form of concurrency *control* is necessary, which ultimately even has to take certain failure situations into account as shown by the dirty-read example. With transactional guarantees, however, all these anomalies should be avoided and the outside world should always get the impression that the executing transactions are run serially, one after the other, in some order.

Another important observation from the three examples is that the details of how data is manipulated within the local variables of the executing programs are mostly irrelevant. Rather it is the interleaved patterns of read and write accesses to the permanently stored data (plus the occurrence of rollbacks on the time-line) that we should focus on. We will see next that these patterns will indeed guide us to a correctness criterion for concurrent transaction executions, in that we can discriminate the allowable patterns from those that could lead to anomalies.

3.3 Syntax of Histories and Schedules

Following the discussion from the previous section, an immediate requirement to a notion of correctness will be that these situations are captured and even avoided. Obviously, we will also be interested in correctness criteria that can be decided *efficiently*, since a *scheduler*, the heart of a concurrency control component, will have to apply them on-line.

As we did for transactions in the previous chapter, we first clarify the *syntax* of schedules. Here we try to respect the dynamic situation in two ways: First, we assume that schedules contain information on how a transaction *ends*. Technically, this can be either *successfully*, indicated by a termination operation called *commit* (c), that is, the transaction in question ran completely and without an interruption by some failure so that the results it has produced can be made visible to other transactions and permanent in the database. Or it may be *unsuccessfully*, which means the transaction has to be terminated prior to reaching its normal end, and which will be indicated by another termination operation called *abort* (a). An aborted transaction should not have any effect on the underlying database, which

termination operations

will mainly be assured by *recovery* procedures to be discussed in Part III. Steps of type c or a will also be called *termination operations*, in order to distinguish them from *data operations* of type r or w (in the page model).

Second, we will make a distinction between schedules in which the outcome of *every* transaction is known, and schedules in which this is open for some or even all transactions. Since in reality the outcome of a set of transactions is actually known *in retrospect* only, we will call schedules of the former type *histories*, and will keep the term "schedule" for transaction interleavings of the second type. In other words, a history will always be *complete* in the sense that all operations of each transaction appear, including a termination operation for every single transaction; note that histories in this sense are also called *complete schedules* in the literature. On the other hand, schedules will in general be prefixes of histories, in which data as well as termination operations of transactions may still be missing. We consider this adequate for capturing a dynamic scheduling situation, since a scheduler will generally know a schedule only as the operations of the various transactions arrive step-by-step, and their execution is normally launched *before* the scheduler has seen the entire transaction. In the ideal situation of continuously error-free operation a schedule (and hence a history) will even become infinitely long.

schedules vs. histories

We have mentioned in Chapter 2 that there are two ways of looking at a transaction: either as a *total* order of steps, in which no parallelism whatsoever occurs, or as a *partial* order of steps, in which some steps may occur (and hence be executed) in parallel. The same distinction is of course appropriate for schedules and histories, as we will discuss in the sequel. We start by defining the general case (partial order) which puts as few restrictions on the ordering of steps on transactions, schedules, or histories as possible.

histories and schedules

Definition 3.4 (Schedules and histories)

Let $T = \{t_1, \ldots, t_n\}$ be a (finite) set of transactions, where each $t_i \in T$ has the form $t_i = (\mathrm{op}_i, <_i)$, with op_i denoting the set of operations of t_i and $<_i$ denoting their ordering, $1 \leq i \leq n$.

(i) A *history* for T is a pair $s = (\mathrm{op}(s), <_s)$ s.t.

(a) $\mathrm{op}(s) \subseteq \bigcup_{i=1}^{n} \mathrm{op}_i \cup \bigcup_{i=1}^{n} \{a_i, c_i\}$, i.e.,
s consists of the union of the operations from the given transactions plus a termination operation, which is either a c_i (commit) or an a_i (abort), for each $t_i \in T$,

(b) $(\forall\, i, 1 \leq i \leq n) \quad c_i \in \mathrm{op}(s) \Longleftrightarrow a_i \notin \mathrm{op}(s)$, i.e.,
for each transaction, there is either a commit or an abort in s, but not both,

(c) $\bigcup_{i=1}^{n} <_i \ \subseteq \ <_s$, i.e.,
all transaction orders are contained in the partial order given by s,

(d) $(\forall\, i, 1 \leq i \leq n)(\forall\, p \in \mathrm{op}_i)\ p <_s a_i$ or $p <_s c_i$, i.e.,
the commit or abort operation always appears as the last step of a transaction, and

(e) any pair of operations $p, q \in \mathrm{op}(s)$ from distinct transactions accessing the same data item s.t. at least one is a write operation is ordered in s in such a way that either $p <_s q$ or $q <_s p$.

(ii) A *schedule* is a prefix of a history.

Thus, a history (for partially ordered transactions) has to contain all operations from all transactions (a), needs a distinct termination operation for every transaction (b), preserves all orders within the transactions (c), has the termination steps as final steps in each transaction (d), and orders "conflicting" operations (e). Because of (a) and (b), a history is also called a "complete schedule" in the literature. The view that two operations which access the same data item and of which at least one is a write operation are in conflict is identical to the notion of conflict that will later be used as the basis for a notion of serializability.

Notice that a prefix of some history can also be the history itself; therefore, histories can be considered a special case of schedules, so that for most considerations that follow, it will suffice to consider some *schedule* to be given.

serial history

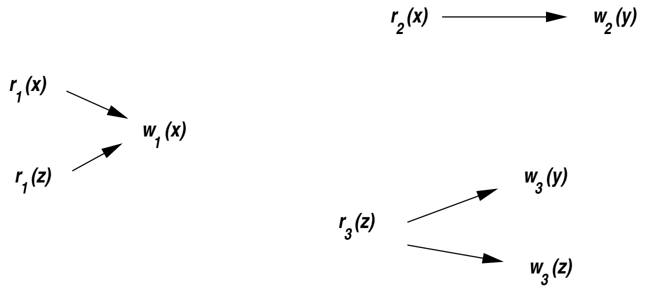

Figure 3.1: Three sample transactions.

Definition 3.5 (Serial history)

A history s is *serial* if for any two transactions t_i and t_j in it, where $i \neq j$, all operations from t_i are ordered in s before all operations from t_j or vice versa.

Example 3.6

Consider the three transactions shown in Figure 3.1. For each transaction, shown as a directed acyclic graph (DAG), the edges exhibit the orderings of steps. Figure 3.2 shows two histories for these transactions, both assuming that each transaction commits. In history (a), the operations are totally ordered as written from left to right. History (b), on the other hand, exhibits a partial order, where the dashed lines are those newly required by Condition (e) in Definition 3.4 (i). □

Note that we often say that a step p in a schedule or history *occurs before* another step q in the same schedule, for which we write "$p < q$;" if the context of a schedule s [transaction t] needs to be expressed, we will analogously write "$p <_s q$" ["$p <_t q$"].

We also note that a prefix of a partial order is essentially obtained by omitting pieces from the end of a "reachability chain." More precisely, if $s = (\mathrm{op}(s), <_s)$, then a *prefix* of s has the form $s' = (\mathrm{op}_{s'}, <_{s'})$ s.t.

(i) $\mathrm{op}_{s'} \subseteq \mathrm{op}(s)$

(ii) $<_{s'} \ \subseteq \ <_s$

(a)

$$r_1(x)r_2(x)r_1(z)w_1(x)w_2(y)r_3(z)w_3(y)c_1c_2w_3(z)c_3$$

(b)

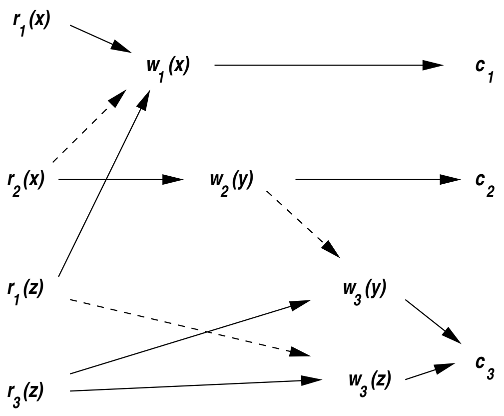

Figure 3.2: Two histories for the transactions from Figure 3.1.

(iii) $(\forall\ p \in \mathrm{op}_{s'})\ (\forall q \in \mathrm{op}(s)) \quad q <_s p \quad \Rightarrow \quad q \in \mathrm{op}_{s'}$

(iv) $(\forall\ p, q \in \mathrm{op}_{s'}) \quad p <_s q \quad \Rightarrow \quad p <_{s'} q$

It is obviously easy to go from a partial order to a total order (by ordering previously unordered steps in an arbitrary way); indeed, partial orders can always be extended to a variety of total orders as special cases. For example, the history shown in Figure 3.2 (a) is a totally ordered one obtained from the history shown in Figure 3.2 (b) by topologically sorting the given (acyclic!) history graph.

shuffle product of transactions

When starting from operation *sequences* or *totally* ordered transactions, however, the appropriate technical tool to define histories and schedules is the *shuffle product*: Let $T = \{t_1, \ldots, t_n\}$ be a (finite) set of totally ordered transactions. shuffle(T) denotes the *shuffle product* of T, i.e., the set of all sequences of steps in which each sequence $t_i \in T$ occurs as a subsequence and which contains no other steps. Notice that this is the shuffle product in an automata-theoretic sense; it is well-defined due to the assumption that distinct transactions do not have steps in common. Formally, the steps of a transaction t_i, $1 \leq i \leq n$, are as before equipped with the index i in an element of a shuffle product.

totally ordered histories and schedules

The notion of a shuffle product allows us to derive a definition of totally ordered histories and schedules for transactions in the page model from Definition 3.4 as follows: Let $T = \{t_1, \ldots, t_n\}$ be a (finite) set of totally ordered transactions. A *history* (or complete schedule) s for T is derived from a sequence $s' \in$ shuffle(T) by adding c_i or a_i for each $t_i \in T$ according to rules (b) and (d) of Definition 3.4 (i) (for simplification, we will refer to this augmentation also simply as the shuffle product). A *schedule* is just as before a prefix of a history, and a history s is *serial* if

$$s = t_{\rho(1)} \ldots t_{\rho(n)}$$

holds for a permutation ρ of $\{1, \ldots, n\}$ (where it is assumed that the termination operations occur last in each transaction).

Example 3.7

Consider the set $T = \{t_1, t_2, t_3\}$ of transactions from Example 3.6 again, which are now perceived as total orders as follows:

$$t_1 = r_1(x)r_1(z)w_1(x)$$
$$t_2 = r_2(x)w_2(y)$$
$$t_3 = r_3(z)w_3(y)w_3(z)$$

The history

$$r_1(x)r_2(x)r_1(z)w_1(x)w_2(y)r_3(z)w_3(y)c_1c_2w_3(z)c_3$$

seen earlier is a totally ordered history and has

$r_1(x)r_2(x)r_1(z)w_1(x)w_2(y)r_3(z)w_3(y)$,
$r_1(x)r_2(x)r_1(z)w_1(x)w_2(y)$, and
$r_1(x)r_2(x)r_1(z)$

among its prefixes. □

Example 3.8

Consider the set $T = \{t_1, t_2, t_3\}$ of transactions, where

$$t_1 = r_1(x)w_1(x)r_1(y)w_1(y),$$
$$t_2 = r_2(z)w_2(x)w_2(z),$$
$$t_3 = r_3(x)r_3(y)w_3(z).$$

Then we have:
$s_1 = r_1(x)r_2(z)r_3(x)w_2(x)w_1(x)r_3(y)r_1(y)w_1(y)w_2(z)w_3(z) \in \text{shuffle}(T)$,
$s_2 = s_1c_1c_2a_3$ is a history in the shuffle product of T augmented by termination steps, where s_1 denotes the sequence of page operations,
$s_3 = r_1(x)r_2(z)r_3(x)$ is a schedule,
$s_4 = s_1c_1$ is another schedule,
$s_5 = t_1c_1t_3a_3t_2c_2$ is serial. □

It is not difficult to verify that the results obtained in this chapter always hold both for total as well as for partial orders. However, they are mostly easier to state for total orders, which is why we will almost without exception go with total orders, in particular when giving examples. We will point to differences where appropriate.

Instead of studying a static situation first and then transferring the results to the dynamic situation actually given in reality, we will develop the theory of serializability here directly for a dynamic setting. Therefore, we will no longer start from a "given" set T of transactions when considering schedules and histories, but we will "read off" the steps of transactions from a given schedule. To this end, we introduce the following notions, intended to ease the distinction between *committed, aborted* and *active* transactions in a given schedule: Let s be a schedule.

transaction sets of a schedule

(i) $\text{trans}(s) := \{t_i \mid s \text{ contains steps from } t_i\}$
trans(s) denotes the set of all transactions occurring partially or completely in s.

(ii) $\text{commit}(s) := \{t_i \in \text{trans}(s) \mid c_i \in s\}$
commit(s) denotes the set of all transactions that are committed in s.

(iii) $\text{abort}(s) := \{t_i \in \text{trans}(s) \mid a_i \in s\}$
abort(s) denotes the set of all transactions that are aborted in s.

(iv) $\text{active}(s) := \text{trans}(s) - (\text{commit}(s) \cup \text{abort}(s))$
active(s) denotes the set of all transactions that are still active in s (i.e., for which termination through commit or abort is still open).

These notations will make it easier to argue about schedules. Consider the previous example again; there we have for schedule

$$s_2 = r_1(x)r_2(z)r_3(x)w_2(x)w_1(x)r_3(y)r_1(y)w_1(y)w_2(z)w_3(z)c_1c_2a_3$$

the following transaction sets:

$$\text{trans}(s_2) = \{t_1, t_2, t_3\}$$
$$\text{commit}(s_2) = \{t_1, t_2\}$$
$$\text{abort}(s_2) = \{t_3\}$$
$$\text{active}(s_2) = \emptyset$$

Similarly, we have for schedule

$$s_4 = r_1(x)r_2(z)r_3(x)w_2(x)w_1(x)r_3(y)r_1(y)w_1(y)w_2(z)w_3(z)c_1$$

the following transaction sets:

$$\text{trans}(s_4) = \{t_1, t_2, t_3\}$$
$$\text{commit}(s_4) = \{t_1\}$$
$$\text{abort}(s_4) = \emptyset$$
$$\text{active}(s_4) = \{t_2, t_3\}$$

As a corollary from the definition of these notions, it should be clear that for each history s,

$$\text{trans}(s) = \text{commit}(s) \cup \text{abort}(s)$$
$$\text{active}(s) = \emptyset$$

In general, for a schedule s the sets commit(s), abort(s), and active(s) are pairwise disjoint and together constitute trans(s). As transaction aborts will mostly be ignored in this chapter, we will almost always look at schedules s s.t. $\text{abort}(s) = \emptyset$.

3.4 Correctness of Histories and Schedules

Our next goal is to devise correctness criteria for schedules. If S denotes the set of all schedules, such a criterion can formally be considered as a mapping

$$\sigma : S \to \{0, 1\}$$

associating a Boolean value with each $s \in S$ (where 0 stands for "false" and 1 for "true"). Thus, *correct* schedules will be those $s \in S$ for which $\sigma(s) = 1$ holds:

$$\text{correct}(S) := \{s \in S \mid \sigma(s) = 1\}$$

A concrete criterion σ should at least meet the following requirements:

1. $\text{correct}(S) \neq \emptyset$, i.e., there are at least some correct schedules in S;

2. "$s \in \text{correct}(S)$" is efficiently decidable, so it does not take a scheduler forever to determine whether what it has come up with is acceptable;

3. $\text{correct}(S)$ is "sufficiently large", so a scheduler based on the respective criterion intuitively has many possibilities to produce correct schedules for the given set of transactions. The larger the set of allowable schedules, the more concurrency will be possible, and thus better performance can be expected.

Our goal in this chapter is to provide several such criteria, under the prerequisite that the transaction semantics is not known. We will work around this by introducing a "syntactical semantics" for schedules which will render it possible to state correctness criteria that are intuitively appropriate. However, our initial approaches will not lead to efficiently decidable criteria, so that restrictions are needed. An important observation, however, is that the way to obtain another criterion will always be the same.

the fundamental idea of serializability

Assuming that schedules should maintain the integrity of the underlying data, i.e., keep it consistent, and assuming that each individual transaction is able to assure this, it is reasonable to conclude (by "induction") that *serial* histories are correct. On the other hand, for performance reasons we are not interested in serial histories, so that we will only use them as a correctness measure, via an appropriately chosen *equivalence relation*. In more detail, we will do the following:

1. We define an equivalence relation "$\approx$" on the set S of all schedules. This gives rise to a decomposition of S into equivalence classes $[\,]$ according to $\approx$:

$$[S]_\approx = \{[s]_\approx \mid s \in S\}$$

   $[S]_\approx$ denotes the set of all equivalence classes according to $\approx$. Clearly, all schedules in *one* such class are pairwise equivalent, so that any schedule s can be chosen as the representative of that class.

2. According to what we said above, the interest will then be in those classes for which a *serial* schedule can be chosen as the representative. Elements of such a class will be called *serializable*.

Thus, we will in brief do the following:

1. Define a notion of equivalence for schedules;

2. define "serializability" via equivalence to serial histories.

We will practice this several times in the remainder of this chapter. We will even practice the same, possibly for a different transaction model, in later chapters, the reason being the simple fact that *every* notion of serializability can indeed be obtained in that way.

aborts disregarded

Throughout this chapter, and also most of the material in the book's Part II, we will largely disregard transaction aborts. Whenever it is easy to incorporate aborts at the "syntactic" level, we will do so, but a full account of their subtle semantics is postponed until Part III. Chapter 11 will study the effects that aborts can have on concurrent executions. So for the rest of this chapter the reader may assume that schedules contain only committed or active transactions.

3.5 Herbrand Semantics of Schedules

We will define equivalence between schedules via a notion of semantics. However, as we have said, this is difficult to make precise in the absence of transaction-program or even application semantics. So what we will do is define a "syntactical semantics" that extends the interpretation of steps of a transaction, introduced in the previous chapter, to the context of a schedule. More precisely, we will first define the semantics of steps occurring in a schedule and then that of the schedule itself. In doing so, we will ignore the fact that transactions may fail, and will hence assume the following for an arbitrary schedule s:

idea of Herbrand semantics

1. A step $r_i(x) \in s$ of a transaction $t_i \in \text{trans}(s)$ reads the value written by the *last* $w_j(x) \in s, j \neq i$, that occurs before $r_i(x)$,

2. a step $w_i(x) \in s$ writes a new value which potentially depends on the values of *all* data items that t_i has read from the database or from other transactions in $\text{active}(s) \cup \text{commit}(s)$ prior to $w_i(x)$.

Notice that the last writer of some data item x is always well-defined in a schedule or history, since situations such as

$$\begin{array}{ccc} w_i(x) & \rightarrow & r_k(x) \\ & \nearrow & \\ w_j(x) & & \end{array}$$

are disallowed by the fact that $w_i(x)$ and $w_j(x)$ must be ordered: If $w_i(x) < w_j(x)$, then $r_k(x)$ reads from $w_j(x)$; if $w_j(x) < w_i(x)$, then $r_k(x)$ reads from $w_i(x)$.

In our exposition of this chapter, we avoid problems that may result from taking aborted (or aborting) transactions into account (see the bibliographic remarks) by assuming that transactions are executed in a failure-free environment, i.e., that each and every transaction eventually commits. We will return to this discussion in particular in Part III of the book, which is on recovery.

For the first assumption introduced above, a problem could arise from the fact that not every read step is preceded by a write step in a schedule, e.g., consider

$$s = r_1(x)r_2(y)w_1(x)r_2(x)\dots.$$

In order to make the result of a read operation well-defined even in such cases we will assume in what follows that every schedule has a fictitious *initializing* transaction t_0 in the beginning, which writes all data items referenced in the schedule (and then commits). For example, the schedule just mentioned thereby becomes

$$s = w_0(x)w_0(y)c_0r_1(x)r_2(y)w_1(x)r_2(x)\dots$$

In words, an initializing transaction defines an initial state to which a given schedule is applied. Clearly, the initializing transaction, which will always be denoted t_0, has to precede all steps from every other transaction occurring in the schedule under consideration. Most of the time, we will not show t_0 explicitly.

The problem with the informal semantics of steps in a schedule given above still is that values read or written, or functions applied, are unknown. One way out of this situation is to use a technique from mathematical logic, namely to define a *Herbrand semantics* by using only uninterpreted function symbols, and nothing else.

Definition 3.9 (Herbrand semantics of steps)

Let s be a schedule. The *Herbrand semantics* H_s of steps $r_i(x), w_i(x) \in$ op(s) is recursively defined as follows:

(i) $H_s(r_i(x)) := H_s(w_j(x))$, where $w_j(x), j \neq i$, is the last write operation on x in s before $r_i(x)$.

(ii) $H_s(w_i(x)) := f_{ix}(H_s(r_i(y_1)), \dots, H_s(r_i(y_m)))$, where the $r_i(y_j), 1 \leq j \leq m$, represent all read operations of t_i that occur in s before $w_i(x)$, and where f_{ix} is an uninterpreted m-ary function symbol.

Note that f_{ix} as used in Part (ii) of the previous definition is well-defined due to our general assumption that there is at most one write step on each data item per transaction. It is then easily verified that $H_s(p)$ is well-defined for each data operation $p \in$ op(s). To this end, it is important that s has been extended by

t_0; every write step of t_0 writing, say, data item x has not read anything before, and will hence be associated with a 0-ary function $f_{0x}()$ providing a constant (the initial value of x). Informally, the previous definition states that in a schedule a read step returns the value of the last write operation executed on behalf of an unaborted transaction (notice that we do not require that only the values produced by committed transactions may be read — this will be the subject of recoverability notions to be introduced in Part III). The result of a write operation depends — in an unknown way — on the values of those data items the writing transaction has previously read.

As an example, the Herbrand semantics H_s of the steps of the schedule

$$s = w_0(x)w_0(y)c_0r_1(x)r_2(y)w_2(x)w_1(y)c_2c_1$$

is as follows, where $f_{0x}()$ and $f_{0y}()$ are 0-ary functions (constants):

$$\begin{aligned}
&H_s(w_0(x)) = f_{0x}()\\
&H_s(w_0(y)) = f_{0y}()\\
&H_s(r_1(x)) = H_s(w_0(x)) = f_{0x}()\\
&H_s(r_2(y)) = H_s(w_0(y)) = f_{0y}()\\
&H_s(w_2(x)) = f_{2x}(H_s(r_2(y))) = f_{2x}(f_{0y}())\\
&H_s(w_1(y)) = f_{1y}(H_s(r_1(x))) = f_{1y}(f_{0x}())
\end{aligned}$$

The example also demonstrates the effect of an initializing transaction, which fixes initial values for all relevant data items.

Next, we will fix a *Herbrand universe* for transactions and, based on that, the semantics of a schedule:

Definition 3.10 (Herbrand universe)

Let $D = \{x, y, z, \dots\}$ be a (finite) set of data items (representing the data of the underlying data server(s)). For a transaction t, let op(t) denote the set of all steps of t. The *Herbrand universe HU* for transactions t_i, $i > 0$, is the smallest set of symbols satisfying the following conditions:

(i) $f_{0x}() \in HU$ for each $x \in D$, where f_{0x} is a 0-ary function symbol (i.e., a constant);

(ii) if $w_i(x) \in$ op(t_i), $|\{r_i(y) \,|\, (\exists\, y \in D)\, r_i(y) <_{t_i} w_i(x)\}| = m$, and if $v_1, \dots, v_m \in HU$, then $f_{ix}(v_1, \dots, v_m) \in HU$, where f_{ix} is an m-ary function symbol.

Thus, the semantics of data operations in a schedule are values from the Herbrand universe. Notice that a Herbrand universe is a purely syntactical construction that is only about symbols, and that does not say anything about the values actually read or written by the transactions in question. Now we have:

semantics of a schedule

Definition 3.11 (Schedule semantics)

The *semantics* of a schedule s is the mapping

$$H[s] : D \to HU$$

defined by

$$H[s](x) := H_s(w_i(x)),$$

where $w_i(x)$ is the last operation from s writing x, for each $x \in D$.

In other words, the semantics of a schedule s is the set of values which are written last in s. Since we are considering a failure-free environment, there is no need to state that only unaborted transactions are taken into account.

As an example, let $s = w_0(x)w_0(y)c_0r_1(x)r_2(y)w_2(x)w_1(y)c_2c_1$ be as above. Then we have:

$$H[s](x) = H_s(w_2(x)) = f_{2x}(f_{0y}())$$
$$H[s](y) = H_s(w_1(y)) = f_{1y}(f_{0x}())$$

The above definitions, although quite general, have the interesting side-effect that they can be adapted to *any* concrete transaction interpretation (see Exercises 2 and 3). Clearly, the price to pay for this generality is the fact that Herbrand semantics is intricate to handle.

3.6 Final-State Serializability

We are now in a position to define a first form of serializability, or equivalence to some *serial* schedule. Basically, such a notion of equivalence will make sense only for *histories*, i.e., for complete schedules, since the serial standard of comparison is by definition complete (all transactions occur completely one after the other). We will later have to discuss the question of how to define correctness for (arbitrary) schedules which are not necessarily histories in our sense.

We start with a first notion of equivalence between schedules which now appears near at hand:

Definition 3.12 (Final-state equivalence)

Let s and s' be schedules. s and s' are called *final-state equivalent*, denoted $s \approx_f s'$, if $\mathrm{op}(s) = \mathrm{op}(s')$ and $H[s] = H[s']$, i.e., s and s' comprise the same set of operations and have the same Herbrand semantics.

Intuitively, two schedules are final-state equivalent if they result in the same final state for any given initial state. For example, consider the following pair of schedules:

$$s = r_1(x)r_2(y)w_1(y)r_3(z)w_3(z)r_2(x)w_2(z)w_1(x)$$
$$s' = r_3(z)w_3(z)r_2(y)r_2(x)w_2(z)r_1(x)w_1(y)w_1(x)$$

Notice that both schedules have the same operations and comprise active transactions only; the initializing transaction t_0 is not shown in both cases. Now we have:

$$H[s](x) = H_s(w_1(x)) = f_{1x}(f_{0x}()) = H_{s'}(w_1(x)) = H[s'](x)$$
$$H[s](y) = H_s(w_1(y)) = f_{1y}(f_{0x}()) = H_{s'}(w_1(y)) = H[s'](y)$$
$$H[s](z) = H_s(w_2(z)) = f_{2z}(f_{0x}(), f_{0y}()) = H_{s'}(w_2(z)) = H[s'](z)$$

Thus, $s \approx_f s'$. Next consider the following two histories:

$$s = r_1(x)r_2(y)w_1(y)w_2(y)c1_1c_2$$
$$s' = r_1(x)w_1(y)r_2(y)w_2(y)c_1c_2$$

Now we have:

$$\begin{array}{rcl} H[s](y) & = & H_s(w_2(y)) = f_{2y}(f_{0y}()) \\ H[s'](y) & = & H_{s'}(w_2(y)) = f_{2y}(H_{s'}(r_2(y))) = f_{2y}(H_{s'}(w_1(y))) \\ & = & f_{2y}(f_{1y}(H_{s'}(r_1(x)))) \\ & = & f_{2y}(f_{1y}(f_{0x}())) \end{array}$$

Thus, $s \not\approx_f s'$. This example in particular shows that final-state equivalence cannot be decided just by looking at final write operations; whatever has preceded these writes must also be taken into account. In particular, in s only the value of y is newly written (by t_2), and this new value is based on the old value of y previously read by t_2. On the other hand, in s' the value of y already written by t_1 influences the final value of y written by t_2, whereas t_1 is without effect in s.

We next formalize these observations; to this end, it is useful to assume that a history s not only has an initializing transaction t_0 in the beginning, but also a new transaction t_∞ in the end which reads all data items mentioned in s (and then commits), i.e., t_∞ reads the new state produced by s, and all steps of t_∞ occur

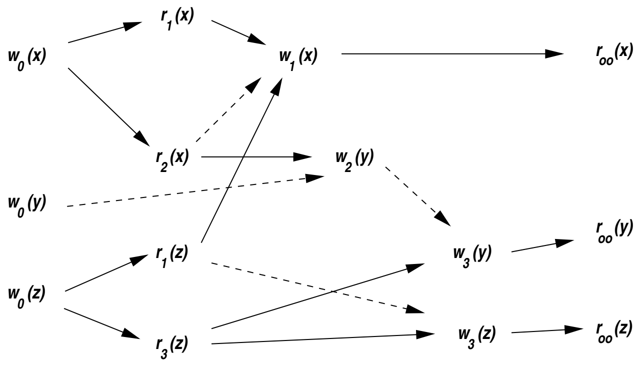

Figure 3.3: A history with an initial and a final transaction.

after all other steps of s. As an example, consider the transactions shown in Figure 3.1 once more. Figure 3.3 shows the history from Figure 3.2 without commits, but completed by initializing and final transactions.

A transaction of type t_∞ does obviously not make sense for a schedule; nevertheless the following notions can be applied to both histories and schedules, for example by ignoring t_∞ or by artificially adding it even to a given (incomplete) schedule.

reads-from relation

Definition 3.13 (Reads-from relation; useful, alive, and dead steps)

Let s be a schedule.

(i) Let $t_j \in \text{trans}(s)$, and let $r_j(x)$ be a read operation of t_j. $r_j(x)$ *reads* x in s *from* $w_i(x), j \neq i$, if $w_i(x)$ is the last write operation on x s.t. $w_i(x) <_s r_j(x)$.

(ii) The *reads-from relation* of s is defined by

$$\text{RF}(s) := \{(t_i, x, t_j) \mid \text{an } r_j(x) \text{ reads } x \text{ from a } w_i(x)\}$$

(iii) A step p is *directly useful* for a step q, denoted $p \to q$, if q reads from p, or if p is a read step and q a subsequent write step from the same transaction.

Let $\stackrel{*}{\to}$ (*"useful"*) denote the reflexive and transitive closure of $\to$.

(iv) A step p is *alive* in s if it is useful for some step from t_∞, i.e.,

$$(\exists\, q \in t_\infty) \quad p \stackrel{*}{\to} q,$$

and *dead* otherwise.

(v) The *live-reads-from relation* of s is defined by

$$\text{LRF}(s) := \{(t_i, x, t_j) \mid \text{an alive } r_j(x) \text{ reads } x \text{ from a } w_i(x)\}.$$

Example 3.14

Consider again the following histories:

$$s = r_1(x) r_2(y) w_1(y) w_2(y) c_1 c_2$$
$$s' = r_1(x) w_1(y) r_2(y) w_2(y) c_1 c_2$$

Then we have:

$$\text{RF}(s) = \{(t_0, x, t_1), (t_0, y, t_2), (t_0, x, t_\infty), (t_2, y, t_\infty)\}$$
$$\text{RF}(s') = \{(t_0, x, t_1), (t_1, y, t_2), (t_0, x, t_\infty), (t_2, y, t_\infty)\}$$

Moreover, in both s and s', $r_2(y) \to w_2(y) \to r_\infty(y)$, i.e., $r_2(y) \stackrel{*}{\to} r_\infty(y)$, and so $r_2(y)$ is alive.

In s', additionally $r_1(x) \to w_1(y) \to r_2(y)$ holds and hence $r_1(x) \stackrel{*}{\to} r_\infty(y)$, that is, $r_1(x)$ is alive. (Note that $r_1(x)$ is not alive in s.) This implies:

$$\text{LRF}(s) = \{(t_0, y, t_2), (t_0, x, t_\infty), (t_2, y, t_\infty)\}$$
$$\text{LRF}(s') = \{(t_0, x, t_1), (t_1, y, t_2), (t_0, x, t_\infty), (t_2, y, t_\infty)\}$$

$\square$

After these considerations, it should no longer come as a surprise that final-state equivalence can alternatively be characterized as follows, which shows that it is easy to test:

Theorem 3.15

Let s and s' be histories. Then

$$s \approx_f s' \quad \text{iff} \quad \text{op}(s) = \text{op}(s') \text{ and } \text{LRF}(s) = \text{LRF}(s').$$

Proof sketch. For a given schedule s we can construct a "step graph" $D(s) = (V, E)$ as follows:

step graph of a schedule

$$V := \text{op}(s)$$
$$E := \{(p, q) \mid p, q \in V, p \to q\}$$

From a step graph $D(s)$ a *reduced* step graph $D_1(s)$ can be derived by dropping all vertices (and their incident edges) which represent dead steps. Then the following can be proven:

1. $\text{LRF}(s) = \text{LRF}(s')$ iff $D_1(s) = D_1(s')$;
2. $s \approx_f s'$ iff $\text{op}(s) = \text{op}(s')$ and $D_1(s) = D_1(s')$.

Clearly, this shows the theorem. □

Example 3.16

The histories

$$s = r_1(x)r_2(y)w_1(y)r_3(z)w_3(z)r_2(x)w_2(z)w_1(x)c_1c_2c_3 \quad \text{and}$$
$$s' = r_3(z)w_3(z)r_2(y)r_2(x)w_2(z)r_1(x)w_1(y)w_1(x)c_3c_2c_1$$

considered earlier have the following LRF relations:

$$\text{LRF}(s) = \{(t_0, x, t_1), (t_0, x, t_2), (t_0, y, t_2), (t_1, x, t_\infty), (t_1, y, t_\infty), (t_2, z, t_\infty)\}$$
$$\text{LRF}(s') = \{(t_0, x, t_2), (t_0, x, t_1), (t_0, y, t_2), (t_1, x, t_\infty), (t_1, y, t_\infty), (t_2, z, t_\infty)\}$$

In both histories, read steps $r_1(x), r_2(y)$, and $r_2(x)$ are alive. Their step graphs $D(s)$ and $D(s')$ are shown in Figure 3.4; the corresponding reduced step graphs are obtained by omitting the circled parts. Clearly, $\text{LRF}(s) = \text{LRF}(s')$ implies $D_1(s) = D_1(s')$ and hence $s \approx_f s'$. □

Example 3.17

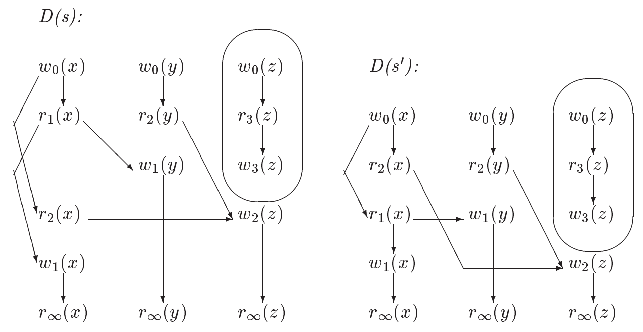

Figure 3.4: (Reduced) step graphs for Example 3.16.

For the histories

$$s = r_1(x)r_2(y)w_1(y)w_2(y)c_1c_2 \quad \text{and}$$
$$s' = r_1(x)w_1(y)r_2(y)w_2(y)c_1c_2$$

from Example 3.14, the step graphs $D(s)$ and $D(s')$ are shown in Figure 3.5, where the reductions are again obtained by deleting the circled parts. Notice that in s only $r_2(y)$ is alive ($(t_0, y, t_2) \in \text{LRF}(s)$); on the other hand, in s' both $r_1(x)$ and $r_2(y)$ are alive ((t_0, x, t_1), $(t_1, y, t_2) \in \text{LRF}(s')$). Therefore, $D_1(s) \neq D_1(s')$ and hence $s \not\approx_f s'$. □

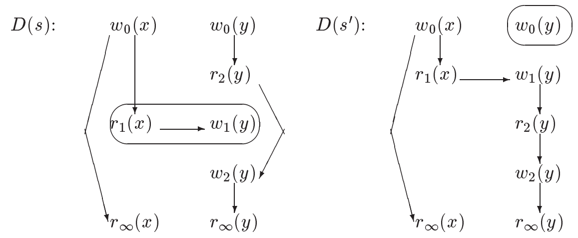

Figure 3.5: (Reduced) step graphs for Example 3.17.

Notice that the step graph of a history is different from the partial order of steps in that history in general. Indeed, in the former, edges are constructed for directly useful steps. For example, if

$$\begin{tikzcd} w_i(x) \arrow[r] & r_k(x) \\ w_j(x) \arrow[ur] & \end{tikzcd}$$

occurs in some given history s, the requirement to order the conflicting writes would yield either $w_i(x) <_s w_j(x)$ or vice versa. Now if $w_i(x) <_s w_j(x)$, there would be no edge from $w_i(x)$ to $r_k(x)$ in $D(s)$, since $r_k(x)$ is forced to read from $w_j(x)$.

The intuition behind a step graph of type $D_1(s)$ for a history s is to restrict the attention to alive steps when testing final-state equivalence, i.e., steps that influence the final state (and are hence on a path to t_∞ in the reduced graph). In light of this interpretation, Theorem 3.15 states that $s \approx_f s'$ holds iff each read step that is alive reads a data item from the same step in both s and s'.

The following is now obvious regarding the complexity of the decision problem asking whether two given schedules are final-state equivalent:

Corollary 3.18

> Final-state equivalence of two schedules s and s' can be decided in time polynomial in the length of the two schedules.

Now given a notion of equivalence, we can define our first notion of serializability:

class FSR

Definition 3.19 (Final-state serializability)

> A history s is *final-state serializable* if there exists a serial history s' s.t. $s \approx_f s'$.
> Let FSR denote the class of all final-state serializable histories.

Example 3.20

> Consider $s = r_1(x)r_2(y)w_1(y)r_3(z)w_3(z)r_2(x)w_2(z)w_1(x)c_1c_2c_3$ from above. Then $s \approx_f t_3t_2t_1$, i.e., $s \in$ FSR. □

The next question to ask naturally is how membership in class FSR can be tested. Clearly, a "brute-force" approach would be to apply Theorem 3.15 to a given history s and — in the worst case — *all* permutations of its transactions; for each such permutation s' we need to test whether $s \approx_f s'$. As we have observed, a *single* such

test is computationally easy (cf. Corollary 3.18); unfortunately, the same may not be expected for the serializability test, since for $|\text{trans}(s)| = n$ there are $n!$ permutations of the n transactions, i.e., exponentially many. This is clearly not efficient, and as we will see shortly there is little hope that a better method exists.

3.7 View Serializability

As we have seen, FSR emphasizes steps that are alive in a schedule. However, since the semantics of a schedule and of the transactions occurring in a schedule are unknown, it is reasonable to require that in two equivalent schedules *each* transaction reads the same values, independent of its liveliness. In this way, it would be assured that each transaction "does" the same in either schedule, or has the same "view" of the underlying data.

This is particularly important due to the fact that in applications, transactions retrieve information from a database and pass this information to their application programs, which in turn may trigger certain real-world actions such as refilling the stock of a product with low quantity on hand. So the retrieved data values are an integral part of the application programs' semantics, regardless of whether the database itself is updated or not.

lost-update and inconsistent-read anomalies reconsidered

The fundamental importance of this observation becomes clear when we reconsider the various types of anomalies that motivate the need for concurrency control, specifically, the lost-update and inconsistent-read canonical examples. These examples now serve as a sanity check for the applicability of the developed theory. Obviously, the lost-update anomaly is indeed properly classified as a case of non-admissible behavior, for the schedule

$$r_1(x)r_2(x)w_1(x)w_2(x)c_1c_2$$

can be shown to violate the FSR criterion. However, the inconsistent-read example reveals that FSR is still insufficient as a correctness criterion, for the schedule

$$r_2(x)w_2(x)r_1(x)r_1(y)r_2(y)w_2(y)c_1c_2$$

would be accepted under the FSR criterion (since the final state is the same as the one produced by t_2t_1) but should actually be disallowed to prevent t_1 from its inconsistent reading. The criterion to be developed next, coined "view serializability", will prevent such inconsistent-read anomalies.

3.7.1 View Equivalence and the Resulting Correctness Criterion

view equivalence

We make the above intuition precise as follows:

Definition 3.21 (View equivalence)

Let s and s' be two schedules. s and s' are called *view equivalent*, denoted $s \approx_v s'$ if the following holds:

(i) $\text{op}(s) = \text{op}(s')$,

(ii) $H[s] = H[s']$,

(iii) $H_s(p) = H_{s'}(p)$ for all read or write steps p.

Conditions (i) and (ii) are just final-state equivalence of s and s'. The third condition additionally requires that each data operation has the same semantics *in either schedule*. Thus, view equivalence is more restrictive than final-state equivalence. View equivalence can be characterized as follows; again, this notion of equivalence is easy to test:

Theorem 3.22

Let s and s' be schedules. The following three statements are pairwise equivalent:

1. $s \approx_v s'$
2. $D(s) = D(s')$
3. $\text{RF}(s) = \text{RF}(s')$

Proof. We here show the equivalence of 1. and 3. and leave the fact that 1. is equivalent to 2. to the reader (see Exercise 5).

"$\Longrightarrow$:" Consider a read step $r_i(x)$ from s. Then

$$H_s(r_i(x)) = H_{s'}(r_i(x)),$$

which implies that if $r_i(x)$ reads x from some step $w_j(x)$ in s, the same holds in s' and vice versa. Since $r_i(x)$ was chosen arbitrarily, it follows that $\text{RF}(s) = \text{RF}(s')$.

"$\Longleftarrow$:" If $\text{RF}(s) = \text{RF}(s')$, this in particular applies to t_∞; hence $H[s] = H[s']$. By the same token, for all other reads $r_i(x)$ in s, we have $H_s(r_i(x)) = H_{s'}(r_i(x))$. Next suppose that for some $w_i(x)$, $H_s(w_i(x)) \neq H_{s'}(w_i(x))$. Thus, the set of values read by t_i prior to step $w_i(x)$ differs in and s and in s', a contradiction to our assumption that $\text{RF}(s) = \text{RF}(s')$. □

Corollary 3.23

View equivalence of two schedules can be tested in time polynomial in the number of elements in $\text{op}(s)$.

With another notion of equivalence, another notion of serializability is now straightforward:

view serializability, class VSR

Definition 3.24 (View serializability)

A history s is *view serializable* if there exists a serial history s' s.t. $s \approx_v s'$.

Let VSR denote the class of all view serializable histories.

It follows from Definition 3.19 that $\text{VSR} \subseteq \text{FSR}$; our next theorem shows that this inclusion is strict:

Theorem 3.25

$\text{VSR} \subset \text{FSR}$

Proof. It suffices to consider the following history:

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

s is final-state equivalent to either one of the two possible serial orders, for one transaction is read-only. However, the schedule is clearly not view serializable, for both serial histories differ from s in terms of the reads-from relation. □

The following theorem, stated here without proof, shows that the difference between FSR and VSR indeed lies in the treatment of steps that are alive:

Theorem 3.26

Let s be a history without dead steps. Then $s \in \text{VSR}$ iff $s \in \text{FSR}$.

Before we turn to the question of how to test membership in VSR, let us again consider from an intuitive level whether VSR is appropriate as a correctness criterion for schedules. If VSR was appropriate, it would have to "filter out" the undesirable situations we have identified for concurrently executed transactions, namely the lost-update and inconsistent-read problems. As we have seen already, a canonical history representing a lost-update situation is

lost update and inconsistent read revisited

$$L = r_1(x)r_2(x)w_1(x)w_2(x)c_1c_2;$$

correspondingly, one representing an inconsistent read is

$$I = r_2(x)w_2(x)r_1(x)r_1(y)r_2(y)w_2(y)c_1c_2$$

Let us look at these histories in turn: For L, there are two possible serial orderings, t_1t_2 and t_2t_1. So what we have to do is compare their RF relations:

$$\begin{aligned}\mathrm{RF}(L) &= \{(t_0,x,t_1),(t_0,x,t_2),(t_2,x,t_\infty)\}\\ \mathrm{RF}(t_1t_2) &= \{(t_0,x,t_1),(t_1,x,t_2),(t_2,x,t_\infty)\}\\ \mathrm{RF}(t_2t_1) &= \{(t_0,x,t_2),(t_2,x,t_1),(t_1,x,t_\infty)\}\end{aligned}$$

Since all RF relations are pairwise distinct, L does not have a view equivalent serial history.

Next we look at history I and have the following:

$$\begin{aligned}\mathrm{RF}(I) = \quad & \{(t_0,x,t_2),(t_0,y,t_2),(t_2,x,t_1),(t_0,y,t_1),\\ & (t_2,x,t_\infty),(t_2,y,t_\infty)\}\\ \mathrm{RF}(t_1t_2) = \quad & \{(t_0,x,t_1),(t_0,y,t_1),(t_0,x,t_2),(t_0,y,t_2),\\ & (t_2,x,t_\infty),(t_2,y,t_\infty)\}\\ \mathrm{RF}(t_2t_1) = \quad & \{(t_0,x,t_2),(t_0,y,t_2),(t_2,x,t_1),(t_2,y,t_1),\\ & (t_2,x,t_\infty),(t_2,y,t_\infty)\}\end{aligned}$$

Again, we find that the RF relations are pairwise different; hence $I \notin$ VSR, as expected.

dirty read revisited

Regarding the third synchronization problem, the dirty-read problem, a canonical history is

$$D = r_1(x)w_1(x)r_2(x)a_1w_2(x)c_2.$$

Since we are ignoring aborted transactions for the time being, there is little we can state about D from the point of view of serializability, since if we omit the aborted t_1, D reduces to $r_2(x)w_2(x)c_2$, for which nothing is wrong so far. We take the fact that D hence seems acceptable under the VSR criterion as an indication that there is something missing in terms of correctness. We will return to this in the book's Part III when we discuss recoverability.

3.7.2 On the Complexity of Testing View Serializability

We now consider the question of how expensive it is to test membership in VSR (and hence in FSR). Of course, the "brute-force approach" described earlier is still applicable: test all possible serial histories for view equivalence to a given history. We will now show that most likely no significantly better approach exists:

Theorem 3.27

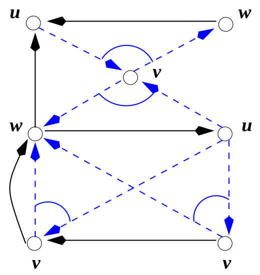

Figure 3.6: A sample polygraph P.

The problem of deciding for a given history s whether $s \in$ VSR holds is NP-complete.

In essence, NP completeness of a problem means that no deterministic algorithm with polynomial running time can solve the problem, unless the universally accepted, albeit unproven, hypothesis "P $\neq$ NP" were disproven.

We will not give the proof of Theorem 3.27 in full detail, since that would take us way beyond the topic of this book; however, we will give sufficient detail on how the proof proceeds. Two steps are the essence:

NP-completeness of VSR testing

1. With a given history, a generalization of directed graphs, a so-called *polygraph*, is associated; for a suitably chosen notion of acyclicity of polygraphs, it can then be shown that a history is in VSR iff its associated polygraph is acyclic. This part of the proof of Theorem 3.27 will be given below.

2. It remains to be shown that testing acyclicity of polygraphs is NP-complete; to this end, we refer the reader to the literature cited in the bibliographic section of this chapter, since this requires a variety of preparations and arguments from complexity theory which would deviate considerably from the subject under discussion here (in particular since, due to this negative complexity result, view serializability will be abandoned as a correctness criterion for transaction histories shortly).

polygraph

A *polygraph* is a triple $P = (V, E, C)$, where (V, E) is a directed graph, and $C \subseteq V \times V \times V$ is a set of *choices* s.t. $(u, v, w) \in C$ implies $u \neq v$, $u \neq w$, $v \neq w$, and $(w, u) \in E$. Figure 3.6 shows a sample polygraph P. In this figure, a choice of the

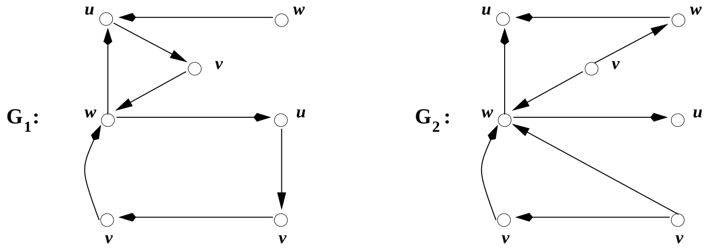

Figure 3.7: Graphs compatible with polygraph P above.

form (u, v, w) is shown as two dashed arrows (u, v) and (v, w) which are connected by a partial circle around v. Note that the figure shows four different choices of the form (u, v, w), which we have not distinguished through indices.

Now let $P = (V, E, C)$ be a polygraph, and let $G = (V, E')$ be a directed graph with the same nodes. G is called *compatible* with P if E' is a minimum set of edges with the properties (where "minimum" refers to set inclusion):

(i) $E \subseteq E'$, i.e., the compatible graph G includes all "normal" edges of the polygraph P, and may have (and usually does have) additional edges,

(ii) $(\forall\ (u, v, w) \in V \times V \times V)\quad (((u, v, w) \in C) \Leftrightarrow ((u, v) \in E' \vee (v, w) \in E'))$, i.e., each choice of the polygraph is represented by an edge in the compatible graph G, and

(iii) $(\forall\ (u, v, w) \in V \times V \times V)\quad (((u, v) \in E' \Leftrightarrow (v, w) \notin E') \wedge ((u, v) \notin E' \Leftrightarrow (v, w) \in E'))$, i.e., each choice is represented by exactly one edge.

In other words, a compatible G "covers" the choices of its underlying polygraph. Now a polygraph P is called *acyclic* if there exists a directed *acyclic* graph (DAG) G that is compatible with P.

For the sample polygraph from Figure 3.6, Figure 3.7 shows two distinct graphs G_1 and G_2 that are compatible with P (where node labels, as before, may stem from different choices). However, while G_1 is cyclic, only G_2, which is acyclic, can be used to verify that the original P is an acyclic polygraph.

Our next goal is to associate a polygraph with a given history. To this end, we will assume for the remainder of this subsection that our history under consideration contains committed transactions only, and is equipped with initializing transaction t_0 and final transaction t_∞. Let s be a history s.t. $\text{trans}(s) = T \cup \{t_0, t_\infty\}$. The

polygraph associated with a history

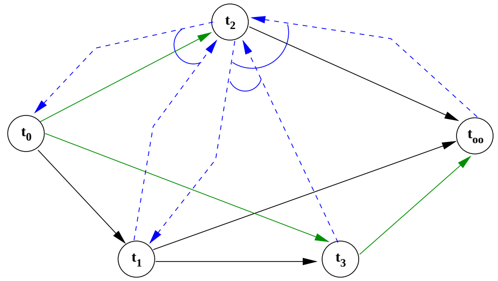

Figure 3.8: A polygraph associated with a history.

polygraph $P(s) = (V, E, C)$ associated with s is defined as follows:

(i) $V = \text{trans}(s)$

(ii) $E = \{(t_0, t) \mid t \in T\} \cup \{(t, t_\infty) \mid t \in T\} \cup \{(t, t') \mid t' \text{ reads some data item from } t\}$

(iii) $C = \{(t', t'', t) \mid (t, t') \in E \wedge\ t' \text{ reads } x \text{ from } t \wedge \text{ some } w(x) \text{ from } t'' \text{ appears somewhere in } s\}$

Note that condition (iii) above implies that the write step of t'', call it $w''(x)$, either precedes the write step of t or follows the read step of t', for t' would not read x from t if $w''(x)$ were in between $w(x)$ and $r'(x)$. The two possible orderings, either $w''(x) <_s w(x)$ or $r'(x) <_s w''(x)$, are captured by the two possible compatible edges that are derivable from the corresponding choices. As an example, consider the following history:

$$s = w_0(x)w_0(y)c_0r_1(x)w_2(y)w_1(y)c_1r_3(y)c_3w_2(x)c_2r_\infty(x)r_\infty(y)c_\infty$$

The associated polygraph $P(s)$ is shown in Figure 3.8. Notice in this figure that some of the solid edges (namely $(t_0, t_2), (t_0, t_3)$, and (t_3, t_∞)) are required only by (part (ii) of) the definition, but are not involved in choices. Also notice that not every edge representing a reads-from relationship gives rise to a choice (e.g., (t_2, t_∞)).

Lemma 3.28

Let s and s' be two histories for the same transactions. If $s \approx_v s'$ then $P(s) = P(s')$.

Proof. From Theorem 3.22, we know that $s \approx_v s'$ iff $D(s) = D(s')$. Now if $D(s)$ is given, the polygraph $P(s)$ can easily be derived:

1. Collapse all nodes from $D(s)$ belonging to the same transaction into one node;
2. create a choice for each edge e of $D(s)$ from t to t' and each transaction $t'' \neq t, t'$ that writes the data item that gave rise to edge e.

Since $D(s) = D(s')$, it follows that $P(s) = P(s')$. □

Lemma 3.29

Let s be a history as before. If s is serial, then $P(s)$ is acyclic.

Proof. Let T be the set of transactions in s, and consider the following graph:

$$G = (T \cup \{t_0, t_\infty\}, E)$$

where

$$E = \{(t, t') \mid t, t' \in T \cup \{t_0, t_\infty\} \wedge t <_s t'\}$$

G thus has edges between any pair of nodes, and since s is assumed to be serial, G must be acyclic.

The claim now is that G is compatible with $P(s)$; if this is true, the lemma follows. First consider an edge (t, t') of $P(s)$; this edge must also be in G since it indicates that t writes a data item later read by t'. Second consider a choice (t, t', t'') of $P(s)$, which indicates that some step p of t' writes a data item that is read by t from t''. Now we need to consider two cases:

(i) If p occurs before the write step from t'' on the data item in question, then p must occur in s before all of t'', since s is serial. Thus, t' occurs in s before t'', and hence (t', t'') is an edge in G.

(ii) p occurs after the read step from t: Now all of t' occurs after all of t in s, which means that (t, t') is in G. □

The following theorem now completes what we want to show here:

Theorem 3.30

For each history s, $s \in$ VSR iff $P(s)$ is acyclic.

Proof. (only if) Let $s \in$ VSR. Then there is some serial history s' s.t. $s \approx_v s'$. By Lemma 3.28, $P(s) = P(s')$; by Lemma 3.29, acyclicity of $P(s')$ implies acyclicity of $P(s)$.

(if) Let $P(s)$ be acyclic, and let G be a DAG compatible with $P(s)$. Note that in G, t_0 is a source node (i.e., a node without incoming edges) and t_∞ is a sink node (i.e., a node without outgoing edges); thus G can be completed to a total ordering G' of the transactions s.t. t_0 appears in the beginning and t_∞ in the end of G'. This total order can be constructed in ascending order by a procedure known as topological sorting. We start out by selecting a node without incoming edges (which is guaranteed to exist in a DAG); if there are several nodes with this property, we can arbitrarily select one of them. Then we remove the selected node and all its outgoing edges from the graph, which renders other nodes to become sources. This simple step is iterated until the graph becomes empty. The order in which the nodes are removed by this procedure is a total order that extends the history's given partial order.

Let s' be a serial history corresponding to G'; we will show that $s \approx_v s'$. Suppose, on the contrary, some step $r(x)$ from t reads from a step $w(x)$ from t' in s, but reads from t'' in s'. Then $P(s)$ contains the choice (t, t'', t'), and G, which is compatible with $P(s)$, contains the edge (t', t); thus, $t' <_{s'} t$. On the other hand, since t reads from t'' in s', it must be the case that $t' < t''$ and $t'' < t$ in s'. Thus, G contains neither the edge (t, t'') nor the edge (t'', t'). However, $P(s)$, as we have said, contains the choice (t, t'', t'), a contradiction to compatibility.

We may therefore conclude that $D(s) = D(s')$ and hence $s \in$ VSR. □

The crucial part is to prove the following theorem:

Theorem 3.31

The problem of deciding whether a given polygraph is acyclic is NP-complete.

As we said, we are not going to prove this here. As with every NP-completeness statement, the proof has two parts: (1) Showing that the problem is in NP is easy; simply guess an "embedded" graph for a given polygraph and verify its compatibility as well as — eventually — its acyclicity. (2) To show completeness, a version of the well-known *satisfiability problem* for Boolean formulas in conjunctive normal form (generally known as SAT) can be reduced to the problem of deciding acyclicity of polygraphs, details of which are beyond the scope of this book.

The bottom line of all this is that, due to the high complexity of its recognition problem, view serializability is inappropriate as a correctness notion for practical scheduling algorithms. However, it turns out that this is not the only reason; to show at least one other, we need the following: Let s be a schedule, $T \subseteq \text{trans}(s)$. $\Pi_T(s)$ denotes the *projection* of s onto T, i.e., the schedule s' with operations $\text{op}(s') = \text{op}(s) - \bigcup_{t \notin T} \text{op}(t)$ obtained by erasing from s all steps not from transactions in T. For example, if

$$s = w_1(x)r_2(x)w_2(y)r_1(y)w_1(y)w_3(x)w_3(y)c_1a_2$$

and $T = \{t_1, t_2\}$. Then

$$\Pi_T(s) = w_1(x)r_2(x)w_2(y)r_1(y)w_1(y)c_1a_2.$$

monotonicity

Definition 3.32 (Monotone classes of histories)

A class E of histories is called *monotone* if the following holds: If s is in E, then $\Pi_T(s)$ is in E for each $T \subseteq \text{trans}(s)$. In other words, E is closed under arbitrary projections.

Monotonicity of a history class E is a desirable property, since it preserves E under arbitrary projections. Taken the other way around, if a projection of some history s does *not* belong to a given class E in a dynamic scheduling situation, then it does not make sense to process s any further (or to extend it with additional operations).

If we now consider the class VSR for E, we can easily verify that VSR is not monotone. Indeed, all we need to exhibit is a counterexample, i.e., a history $s \in$ VSR s.t. for some $T \subseteq \text{trans}(s)$, $\Pi_T(s) \notin$ VSR. To this end, consider

$$s = w_1(x)w_2(x)w_2(y)c_2w_1(y)c_1w_3(x)w_3(y)c_3.$$

Then $s \approx_v t_1t_2t_3 \approx_v t_2t_1t_3$. Thus, $s \in$ VSR. However, $\Pi_{\{t_1,t_2\}}(s) \notin$ VSR. Intuitively, in this sample history the last transaction t_3 "covers" whatever effects have previously been created by the (interleaved) transactions t_1 and t_2; hence, if t_3 disappears by way of a projection, the correction it brings along does not apply anymore.

So in conclusion, VSR as covered here so far is insufficient as a correctness criterion for histories, which is why further restrictions are needed, to be introduced next.

3.8 Conflict Serializability

Our next notion of serializability, *conflict serializability*, is the one which is most important for the practice of transactional information systems, in particular for

building schedulers. As will be seen, it is computationally easy to test and thus differs significantly from the notions of serializability discussed so far. In addition, conflict serializability has a number of interesting theoretical properties which can justify an exploitation of this concept in practice in a variety of ways. Finally, it can be generalized to other transaction models and different data settings, as will be demonstrated in subsequent chapters.

3.8.1 Conflict Relations

Conflict serializability is based on a simple notion of conflict which was mentioned already in connection with partially ordered histories, and which is appropriate for the syntactical nature of read/write transactions:

conflicts

Definition 3.33 (Conflicts and conflict relations)

Let s be a schedule, $t, t' \in \text{trans}(s)$, $t \neq t'$:

(i) Two data operations $p \in t$ and $q \in t'$ are *in conflict* in s if they access the same data item and and at least one of them is a write, i.e.,

$$(p = r(x) \wedge q = w(x)) \vee (p = w(x) \wedge q = r(x)) \vee (p = w(x) \wedge q = w(x))$$

(ii) $\text{conf}(s) := \{(p,q) \mid p, q \text{ are in conflict in } s \text{ and } p <_s q\}$ is called the *conflict relation* of s.

Naturally, only data operations can be in conflict in a schedule; note that the conflict relation of a schedule does not take the termination status of a transaction (i.e., committed or aborted) into account. If aborted transactions were present in a given schedule s, they could, however, be simply ignored, and their conflicts with other transactions omitted from conf(s). For example, if

$$s = w_1(x)r_2(x)w_2(y)r_1(y)w_1(y)w_3(x)w_3(y)c_1a_2,$$

where t_1 is committed, t_2 is aborted, and t_3 is still active, we have

$$\text{conf}(s) = \{(w_1(x), w_3(x)), (r_1(y), w_3(y)), (w_1(y), w_3(y))\}.$$

Now we are ready to introduce the following notion of equivalence:

conflict equivalence

Definition 3.34 (Conflict equivalence)

Let s and s' be two schedules. s and s' are called *conflict equivalent*, denoted $s \approx_c s'$, if they have the same operations and the same conflict relations, i.e., if the following holds:

(i) $\mathrm{op}(s) = \mathrm{op}(s')$ and

(ii) $\mathrm{conf}(s) = \mathrm{conf}(s')$.

Thus, two schedules are conflict equivalent if all conflicting pairs of steps from distinct transactions occur in the same order in both schedules. As an example, let

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

and

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

Then $\mathrm{op}(s) = \mathrm{op}(s')$ and $\mathrm{conf}(s) = \mathrm{conf}(s')$; therefore, $s \approx_c s'$.

Clearly, the notion of conflict equivalence is easy to test: For two given schedules over the same set of operations, derive their conf-relations and test them for equality. Similarly to view equivalence, this can be described in terms of graphs: Let s be a schedule, and let a graph $D_2(s) := (V, E)$ be defined by $V = \mathrm{op}(s)$ and $E = \mathrm{conf}(s)$. Graph $D_2(s)$ is called the *conflicting-step graph* of schedule s. Then we can prove that

conflicting-step graph

$$s \approx_c s' \iff D_2(s) = D_2(s').$$

Notice that if s is given as a partially ordered schedule, $D_2(s)$ is essentially derived by considering the edges that represent inter-transactional conflicts (the dashed edges in Figure 3.2(b)).

3.8.2 Class CSR

We are now ready to introduce a third notion of serializability:

conflict serializability, class CSR

Definition 3.35 (Conflict serializability)

A history s is *conflict serializable* if there exists a serial history s' s.t. $s \approx_c s'$.

Let CSR denote the class of all conflict serializable histories.

For a first example, history

$$r_1(x)r_2(x)r_1(z)w_1(x)w_2(y)r_3(z)w_3(y)c_1c_2w_3(z)c_3$$

from Figure 3.2(a) is in CSR. Next, for history $s = r_2(y)w_1(y)w_1(x)c_1w_2(x)c_2$, we have that $s \notin \mathrm{CSR}$. For $s' = r_1(x)r_2(x)w_2(y)c_2w_1(x)c_1$ we find $s' \in \mathrm{CSR}$.

Next let us briefly check again on the (undesirable) histories L and I seen in the lost-update and inconsistent-read examples, where

lost update and inconsistent read revisited

$$L = r_1(x)r_2(x)w_1(x)w_2(x)c_1c_2$$

and

$$I = r_2(x)w_2(x)r_1(x)r_1(y)r_2(y)w_2(y)c_1c_2.$$

For L, we have

$$\mathrm{conf}(L) = \{(r_1(x), w_2(x)), (r_2(x), w_1(x)), (w_1(x)w_2(x))\}.$$

Now the only possible serial histories are t_1t_2, whose conflict relation would avoid the second pair of conflicting operations, and t_2t_1, whose conflict relation would comprise the second pair only. Thus, L cannot be conflict serializable. Next, for I we find

$$\mathrm{conf}(I) = \{(w_2(x), r_1(x)), (r_1(y), w_2(y))\},$$

which again cannot be obtained from t_1t_2 or from t_2t_1. Thus, $I \notin \mathrm{CSR}$. In other words, conflict serializability avoids lost updates and inconsistent reads.

We next clarify the relationship between class CSR and the classes of serializable histories defined earlier:

Theorem 3.36

$\mathrm{CSR} \subset \mathrm{VSR}$

The fact that CSR is a subset of VSR follows from the fact that graph $D(s)$ is uniquely determined by $D_2(s)$, provided the set of edges of $D_2(s)$ represents the conflict relation $\mathrm{conf}(s)$ of s (in the sense of Definition 3.33).

Another way to show this is as follows: Consider an $s \in \mathrm{CSR}$. By definition, there is a serial history s' s.t. $s \approx_c s'$. We show that $\mathrm{RF}(s) = \mathrm{RF}(s')$. Suppose this is not the case, and let (t_i, x, t_j) be in $\mathrm{RF}(s)$, but not in $\mathrm{RF}(s')$. Then an $r_j(x)$ reads x from some $w_i(x)$ in s (and $(w_i(x), r_j(x)) \in \mathrm{conf}(s)$), but $r_j(x)$ reads x from some other $w_k(x)$, $k \neq i$, in s' (and $(w_k(x), r_j(x)) \in \mathrm{conf}(s')$). Clearly, $w_i(x)$ and $w_k(x)$ are also in conflict, and we find $w_i(x) < w_k(x)$ in s', but $w_k(x) < w_i(x)$ in s, a contradiction to the fact that $s \approx_c s'$ (and hence the two histories have the same conflict relations). Therefore, $\mathrm{RF}(s) = \mathrm{RF}(s')$, and $s \approx_v s'$, which implies $s \in \mathrm{VSR}$.

To see that the inclusion is strict, consider the following history:

$$s = w_1(x)w_2(x)w_2(y)c_2w_1(y)c_1w_3(x)w_3(y)c_3$$

Then $s \notin$ CSR, but $s \approx_v t_1t_2t_3$, so $s \in$ VSR. □

Corollary 3.37

CSR ⊂ VSR ⊂ FSR

The following can also be verified:

Theorem 3.38

(i) The class CSR is monotone.
(ii) $s \in \text{CSR} \iff (\forall\, T \subseteq \text{trans}(s))\ \Pi_T(s) \in \text{VSR}$
(i.e., CSR is the largest monotone subset of VSR).

Clearly, Part (i) of this theorem is straightforward, whereas Part (ii) is not; we leave it to the reader as an admittedly challenging exercise.

The important difference between CSR and the previously introduced classes FSR and VSR now is that membership in the former can be tested efficiently. To this end, we will characterize conflict serializability in graph-theoretic terms via the *conflict graph* associated with a history; in this graph, nodes represent committed transactions, and an edge from transaction t to transaction t' indicates that there are steps $p \in t$ and $q \in t'$ s.t. p and q are in conflict.

conflict graph (serializaton graph) of a schedule

As a motivation why we can test conflict serializability by using a properly defined graph, consider history

$$s = r_1(y)r_3(w)r_2(y)w_1(y)w_1(x)w_2(x)w_2(z)w_3(x)c_1c_2.$$

Intuitively, s cannot be conflict equivalent to a serial history, since "$r_2(y)w_1(y)$" says "$t_2 < t_1$ in an equivalent serial history," while "$w_1(x)w_2(x)$" means "$t_1 < t_2$;" obviously, this is impossible to meet simultaneously. The conflict graph of s would represent each transaction as a node, and each conflict as a (directed) edge that respects the ordering of the conflicting steps. The result would be a cycle in the graph, which indicates non-serializability.

Definition 3.39 (Conflict graph (serialization graph))

Let s be a schedule. The *conflict graph*, also known as the *serialization graph*, $G(s) = (V, E)$ of s, is defined by

$$V = \text{commit}(s),$$
$$(t, t') \in E \iff t \neq t' \wedge (\exists\, p \in t)(\exists\, q \in t')\ \ (p, q) \in \text{conf}(s)$$

The conflict graph $G(s)$ of a schedule s thus abstracts from individual conflicts between pairs of transactions and represents multiple conflicts between the same (committed) transactions by a single edge. In this sense it contains less information than the graph $D_2(s)$ defined earlier; as will be seen shortly, however, this information suffices for testing membership in CSR.

Example 3.40

Consider

$$s = r_1(x)r_2(x)w_1(x)r_3(x)w_3(x)w_2(y)c_3c_2w_1(y)c_1.$$

The conflict graph $G(s)$ is shown in Figure 3.9. □

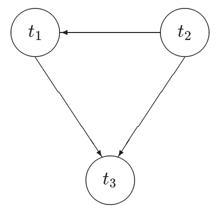

Figure 3.9: Conflict graph $G(s)$ for Example 3.40.

The following is widely known as the "(conflict) serializability theorem:"

Theorem 3.41

Let s be a history. Then

$s \in$ CSR iff $G(s)$ is acyclic.

Proof. (only if) Let $s \in$ CSR, then there exists a serial history s' s.t. $\text{op}(s) = \text{op}(s')$ and $\text{conf}(s) = \text{conf}(s')$, i.e., $s \approx_c s'$. Now consider $t, t' \in V$, $t \neq t'$, with $(t, t') \in E$ for $G(s) = (V, E)$. Then we have:

$$(\exists\, p \in t)(\exists\, q \in t') \quad p <_s q \text{ and } (p, q) \in \text{conf}(s).$$

Since the conflict relations of s and s' are equal, $p <_{s'} q$. On the other hand, since s' is serial, all of t must hence occur before all of t' in s'.

Now suppose $G(s)$ was cyclic. Then there must be a cycle of the form, say, $t_1 \to t_2 \to \ldots \to t_k \to t_1$. The same cycle also exists in $G(s')$, a contradiction to the fact that s' is serial.

(if) Conversely, let $G(s) = (V, E)$, where $V = \{t_1, \ldots, t_n\}$, be acyclic. Then $G(s)$ can be topologically sorted to derive a total order of the nodes that includes partial order E; call the result $t_{\rho(1)} \ldots t_{\rho(n)} = s'$ for some permutation ρ of $\{1, \ldots, n\}$. Clearly, s' is a serial history. We now show $s \approx_c s'$: If $p \in t$ and $q \in t'$ for $t, t' \in V$ s.t. $p <_s q$ and $(p, q) \in \text{conf}(s)$, then $(t, t') \in E$. It follows that t is also before t' in the topological sort of s, i.e., in s'. Since p and q are still in conflict, we conclude $\text{conf}(s) \subseteq \text{conf}(s')$. Conversely, if $(p, q) \in \text{conf}(s')$ and $p \in t$, $q \in t'$, then $p <_s q$ and hence $(p, q) \in \text{conf}(s)$. Thus, $\text{conf}(s) = \text{conf}(s')$, and the theorem follows. □

Corollary 3.42

Membership in CSR can be tested in time polynomial in the number of transactions in the given schedule.

By the serializability theorem, the latter is due to the fact that the conflict graph of a given history can be constructed in time linear in the number of operations in the history, and this graph can be tested for cycles in time that is at most quadratic in the number of nodes.

Example 3.43

Consider the following histories:

$s = r_1(y) r_3(w) r_2(y) w_1(y) w_1(x) w_2(x) w_2(z) w_3(x) c_1 c_3 c_2$ and
$s' = r_1(x) r_2(x) w_2(y) w_1(x) c_2 c_1$

The corresponding conflicts graphs $G(s)$ and $G(s')$ are shown in Figure 3.10. Since $G(s)$ contains a cycle, $s \notin$ CSR; since $G(s')$ is acyclic, $s' \in$ CSR. □

Let us mention that "serializability theorems" in the style of Theorem 3.41 occur frequently in the literature. Indeed, generalizations of the read/write transaction model, or completely different transaction models, can also give rise to a graph-theoretic characterization of their correctness criterion (or at least of one of their criteria), so that a theorem analogous to the previous one is obtained. We will see several such theorems in the remainder of this book.

3.8.3 Conflicts and Commutativity

We next characterize conflict serializability in a different way. Our presentation initially considers only totally ordered schedules since these are closer to the "alge-

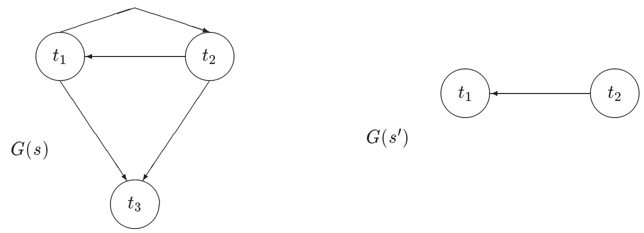

Figure 3.10: Conflict graphs for Example 3.43.

braic" nature of the following *commutativity rules* for page-model data operations. In these rules "$\sim$" means that the ordered pair of actions on the left-hand side can be replaced by the right-hand side, and vice versa.

commutativity rules

Rule C1: $r_i(x) r_j(y) \sim r_j(y) r_i(x)$ if $i \neq j$
Rule C2: $r_i(x) w_j(y) \sim w_j(y) r_i(x)$ if $i \neq j, x \neq y$
Rule C3: $w_i(x) w_j(y) \sim w_j(y) w_i(x)$ if $i \neq j, x \neq y$

Rule C1 says that two read steps $r_i(x)$ and $r_j(y), i \neq j$, which occur in a schedule in this order and are adjacent (with no other operation in between), may be commuted. Similarly, C2 says that a read and write step can be exchanged if they are from distinct transactions and access different data items; finally, C3 says that two write steps can be commuted if they refer to different data items.

The commutativity rules can be applied to a given schedule or history in a stepwise fashion, as in the following example:

$$\begin{aligned} s &= w_1(x) \underbrace{r_2(x) w_1(y)} w_1(z) \underbrace{r_3(z) w_2(y)} w_3(y) w_3(z) \\ &\overset{C2}{\Rightarrow} w_1(x) w_1(y) \underbrace{r_2(x) w_1(z)} w_2(y) r_3(z) w_3(y) w_3(z) \\ &\overset{C2}{\Rightarrow} w_1(x) w_1(y) w_1(z) r_2(x) w_2(y) r_3(z) w_3(y) w_3(z) \\ &\equiv t_1 t_2 t_3 \end{aligned}$$

The above transformations have implicitly assumed that operations in a schedule are totally ordered. With partial orders, we may need an additional transformation rule as an auxiliary step for applying one of the commutativity rules. This additional rule simply states that two unordered operations can be arbitrarily ordered if they are non-conflicting. We refer to this rule as the *ordering rule*:

Rule C4: $o_i(x), p_j(y)$ *unordered*
$\rightsquigarrow o_i(x)p_j(y)$ if $x \neq y \lor (o = r \land p = r)$

ordering rule

We next use the commutativity rules for introducing another relation on schedules:

Definition 3.44 (Commutativity-based equivalence)

Let s and s' be two schedules s.t. $\text{op}(s) = \text{op}(s')$. Define $s \sim s'$ if s' can be obtained from s by a single application of C1, C2, C3, or C4 to the steps of the schedule.

Let "$\overset{*}{\sim}$" denote the reflexive and transitive closure of "$\sim$", i.e., $s \overset{*}{\sim} s'$ if s' can be obtained from s by a finite number of applications of C1, C2, C3, and C4.

It is straightforward to verify that "$\overset{*}{\sim}$" is an equivalence relation on the set of all schedules for a given set of transactions.

Theorem 3.45

Let s and s' be schedules s.t. $\text{op}(s) = \text{op}(s')$. Then

$$s \approx_c s' \text{ iff } s \overset{*}{\sim} s'$$

We leave the proof of this theorem to the reader as an exercise. A first exploitation of these commutativity rules now is that finitely many applications of the rules may transform a given history into a serial one as in the example we have seen.

Definition 3.46 (Commutativity-based reducibility)

A history s is *commutativity-based reducible* if there is a serial history s' s.t. $s \overset{*}{\sim} s'$, i.e., s can be transformed into s' through a finite number of allowed transformation steps according to rules C1, C2, C3, and C4.

Corollary 3.47

A history s is commutativity-based reducible iff $s \in \text{CSR}$.

An important application of the alternative characterization of conflict equivalence given above is that it can immediately be generalized. Indeed, it is completely irrelevant to know of a schedule whether a step reads or writes or which data item it accesses, as long as we know which steps of the schedule are in conflict. The latter suffices for deciding about the correctness of the schedules, even without any knowledge about the meaning of the operations.

To see this, consider the following schedule, whose steps are not specified any further:

$$s = p_1 q_1 p_2 o_1 p_3 q_2 o_2 o_3 p_4 o_4 q_3$$

Suppose s is a schedule for transactions t_1, t_2, and t_3, and assume we know that the following pairs of steps are in conflict:

1. (q_1, p_2),
2. (p_2, o_1),
3. (q_1, o_1),
4. (o_4, q_3).

As before, we can now establish a conflict graph for this schedule (assuming that all transactions are unaborted) and then see that s is *not* conflict serializable, since that graph has a cycle involving transactions t_2 and t_3.

The important point here is that the steps of a schedule or history to which an argument like the above is applied may be of a completely different type, such as *increment* and *decrement* on a counter object, *push* and *pop* on a stack object, or *enqueue* and *dequeue* on a queue object. We will later show how to exploit this observation for coming up with notions of serializability that are based on semantic information. In essence it suffices to state a conflict or commutativity table for "new" operations and derive conflict serializability from that table.

3.8.4 Restrictions of Conflict Serializability

We next look into several restrictions of conflict serializability that are important in practical applications.

3.8.4.1 Order-Preserving Conflict Serializability

We first discuss a restriction of conflict serializability, *order-preserving* conflict serializability, which requires that transactions that do not overlap in time (i.e., that occur strictly sequentially in a given schedule) appear in the same order in a conflict equivalent schedule.

Example 3.48

Consider $s = w_1(x)r_2(x)c_2w_3(y)c_3w_1(y)c_1$. Figure 3.11 shows the conflict graph of s; since it is acyclic, it follows that $s \in$ CSR. Now observe that the equivalent serial history is $t_3t_1t_2$. However, t_2 is already committed in s before t_3 starts. This contrast between the serialization order and the actual execution order could be undesirable; for example, if a user wants transactions executed in the order they are submitted to the system, he or she may be waiting for transactions to complete before submitting the next one. □

order preservation

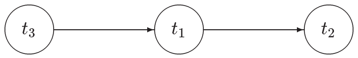

Figure 3.11: Conflict graph for Example 3.48.

A situation like the one just described is avoided by the following criterion:

order-preserving conflict serializability

Definition 3.49 (Order preservation)

A history s is called *order-preserving conflict serializable*, if it is conflict serializable, i.e., there exists a serial history s' s.t. $\mathrm{op}(s) = \mathrm{op}(s')$ and $s \approx_c s'$, and if the following holds for all $t, t' \in \mathrm{trans}(s)$: If t occurs completely before t' in s, then the same holds in s'.
Let OCSR denote the class of all order-preserving conflict serializable histories.

The following is straightforward:

Theorem 3.50

OCSR $\subset$ CSR

Indeed, OCSR $\subseteq$ CSR follows from the last definition, and the sample history s from the previous example shows that the inclusion is strict: $s \in$ CSR − OCSR.

3.8.4.2 Commitment Ordering

We next present another restriction of class CSR which will turn out useful for distributed, possibly heterogeneous, environments as well. It is based on the observation that for transactions in conflict, an ordering of their commits in "conflict order" is sufficient for conflict serializability.

commit-order-preserving conflict serializability

Definition 3.51 (Commit-order preservation)

Let s be a history. s is *commit-order-preserving conflict serializable* (or has the property of *commit-order preservation*), if the following holds: For all transactions $t_i, t_j \in \mathrm{commit}(s)$, $i \neq j$, if $(p, q) \in \mathrm{conf}(s)$ for $p \in t_i, q \in t_j$, then $c_i <_s c_j$ in s, i.e., for committed transactions the ordering of operations in conflict determines the ordering of their commit operations.

Let COCSR denote the class of all histories that are commit-order-preserving conflict serializable.

The following result is a consequence of the serializability theorem:

Theorem 3.52

COCSR $\subset$ CSR

Proof. Let $s \in$ COCSR and (t_i, t_j) be an edge in the conflict graph $G(s)$. By the definition of COCSR, we may then conclude that $c_i <_s c_j$ (due to the conflict between the two transactions). By induction, we can generalize this statement from edges to paths in $G(s)$ (details are omitted), i.e., if $(t_1, \ldots, t_n)$ is a path in $G(s)$ from t_1 to t_n, then $c_1 <_s c_2$, $c_2 <_s c_3$, …, $c_{n-1} <_s c_n$.

Now suppose $s \notin$ CSR. Then, by the serializability theorem, $G(s)$ has a cycle. Without loss of generality, let that cycle have the form $(t_1, \ldots, t_n, t_1)$. By the above, we may then conclude that $c_1 <_s c_1$, a contradiction.

To see that the inclusion is strict, consider the history $s = r_1(x)w_2(x)c_2c_1$, for which $s \in$ CSR–COCSR holds. □

Our next theorem essentially states that a history is commit-order preserving iff the ordering of the commits corresponds to some serialization order:

Theorem 3.53

Let s be a history. $s \in$ COCSR iff
(i) $s \in$ CSR and
(ii) there exists a serial s' s.t. $s' \approx_c s$ and for all $t, t' \in$ trans(s),
$t <_{s'} t' \quad \Rightarrow \quad c_t <_s c_{t'}$.

Proof. "$\Longrightarrow$:" Let $s \in$ COCSR. By Theorem 3.52, $s \in$ CSR. Thus there is a serial history s' s.t. $s' \approx_c s$. Now consider transactions $t, t' \in \text{trans}(s) = \text{trans}(s')$ s.t. $t <_{s'} t'$. Since s and s' are conflict equivalent, it follows that if operations from t and t', say, p and q, are in conflict in s, then $p <_s q$. Due to the COCSR property, this implies $c_t <_s c_{t'}$.

"$\Longleftarrow$:" Assume that $s \notin$ COCSR. Thus, for two transactions $t_i, t_j \in \text{commit}(s)$ there are operations $p \in t_i$ and $q \in t_j$ s.t. $(p, q) \in \text{conf}(s)$, but $c_j < c_i$. Since $s \in$ CSR, let s' be the conflict equivalent serial schedule; due to the conflict between t_i and t_j, $t_i <_{s'} t_j$. However, by our assumption this implies $c_i <_s c_j$, a contradiction. □

We next investigate the relationship between OCSR and COCSR; to this end it turns out that COCSR is even more restrictive than OCSR:

Theorem 3.54

COCSR $\subset$ OCSR

Proof. Let $s \in$ COCSR, and suppose $s \notin$ OCSR. Then there exists a serial history s' s.t. $s \approx_c s'$, but for two transactions $t_i, t_j \in \text{trans}(s)$, $i \neq j$, we have that $t_i <_s t_j$, while $t_i \not<_{s'} t_j$. From the assumption that s is in COCSR we know that $t_i <_s t_j$ implies $c_i <_s c_j$. Now we need to consider two cases:

(i) t_i and t_j are in conflict, i.e., there are two data operations p from t_i and q from t_j s.t. $(p, q) \in \text{conf}(s)$. Since $s \approx_c s'$, it follows that $(p, q) \in \text{conf}(s')$, which implies $t_i <_{s'} t_j$, since s' is serial, but this contradicts our finding above.

(ii) If t_i and t_j are not in conflict, then our assumption that t_i and t_j are reordered in s' must be due to a third transaction t_k, $k \neq i, j$, s.t.

(a) t_k is in conflict with t_i, implying $c_k <_s c_i$, and

(b) t_j is in conflict with t_k, implying $c_j <_s c_k$,

both by the COCSR property. Thus, $c_j <_s c_k <_s c_i$, again a contradiction. □

The fact that the inclusion is strict is implied by Figure 3.12, which shows the relationship between classes CSR, OCSR, and COCSR, and in which the histories mentioned are as follows:

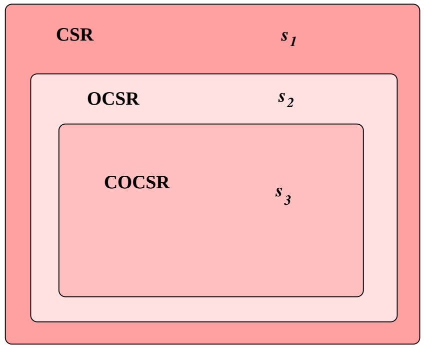

Figure 3.12: Relationship between classes CSR, OCSR, and COCSR.

$$s_1 = w_1(x)r_2(x)c_2w_3(y)c_3w_1(y)c_1 \in \text{CSR} - \text{OCSR}$$
$$s_2 = w_3(y)c_3w_1(x)r_2(x)c_2w_1(y)c_1 \in \text{OCSR} - \text{COCSR}$$
$$s_3 = w_3(y)c_3w_1(x)r_2(x)w_1(y)c_1c_2 \in \text{COCSR}$$

3.9 Commit Serializability

Following the discussion in Section 3.8 it should be clear that conflict serializability is a reasonable starting point for a correctness criterion for histories even in practical applications. However, conflict serializability alone is certainly not enough since, for example, the dirty-read problem again remains undetected. In this section, we will present further observations of this kind, and will provide corresponding "corrections" for the deficiencies we will recognize on the serializability notions we have already established.

The choice of serializability as a correctness criterion for histories is partially based on the idealized assumption that transactions are processed in a failure-free environment so that each individual transaction always reaches a point of successful termination, and that additionally the system never crashes. In reality, however, this is not always the case, since transactions can abort and the system can be interrupted at any point. For these reasons, the following aspects are relevant to our discussion (and need consideration):

1. Since each transaction that is still active in a schedule can abort in the future

(because it is not yet committed), and since a system crash can even force every active transaction to abort, a correctness criterion should actually take only the committed transactions into account.

2. Since a system crash can stop the processing of a schedule at any point, for a correct schedule each of its prefixes should also be correct.

We will now formalize these requirements as *closure properties of schedule properties*, an approach we have encountered already when discussing monotonicity. To this end, we will use schedule projections as introduced earlier in this chapter. Later we will often refer specifically to the projection of a schedule s onto the set commit(s) of its committed transactions, i.e., to $\Pi_{\text{commit}(s)}(s)$; therefore, we will use the shorthand notation "CP(s)" ("*committed projection* of s") for this particular projection.

committed projection of a schedule

Definition 3.55 (Closure properties of schedules properties)

Let E be a class of schedules.

(i) E is *prefix-closed* if
for every schedule s in E each prefix of s is also in E.

(ii) E is *commit-closed* if
for every schedule s in E CP(s) is also in E.

Now requirements 1 and 2 above translate into the requirement that a correctness criterion for schedules that are processed in a not necessarily failure-free environment should exhibit *both* closure properties; in other words, it should be *prefix-commit-closed*. It follows that if a schedule class E is prefix-commit-closed, we have that if E holds for some schedule s, then E also holds for CP(s'), where s' is an arbitrary prefix of s.

prefix-commit closedness

The concrete schedule properties we will look at are membership in one of the classes FSR, VSR, and CSR; thus, we investigate whether membership in one of these classes is prefix-commit-closed. To start with, we have the negative result that membership in classes FSR and VSR is not prefix-commit-closed. For VSR, this follows immediately from the fact that membership in VSR is not monotone; hence it cannot be prefix-closed. For FSR, the following schedule is a counterexample:

$$s = w_1(x)w_2(x)w_2(y)c_2w_1(y)c_1w_3(x)w_3(y)c_3.$$

As is easily verified, $s \approx_v t_1t_2t_3$. Thus, $s \in$ VSR and hence $s \in$ FSR. Now consider the prefix $s' = w_1(x)w_2(x)w_2(y)c_2w_1(y)c_1$ of s. Clearly, CP(s') $= s'$; in addition,

$s' \not\approx_f t_1t_2$ as well as $s' \not\approx_f t_2t_1$. Thus, $s' \notin$ FSR. In other words, we have identified a history $s \in$ FSR s.t. CP(s') ($= s'$) $\notin$ FSR for some of its prefixes s'. This proves the claim.

As we have already recognized FSR and VSR as unsuited for our purposes for reasons of complexity, this is not a "tragic" result. Fortunately, the following holds for CSR:

Theorem 3.56

Membership in class CSR is prefix-commit-closed.

Proof. Let $s \in$ CSR. Then $G(s)$ is acyclic. For each restriction s' of s, $G(s')$ then is a subgraph of $G(s)$ and hence acyclic as well. In particular, for each prefix s' of s, the conflict graph of the projection onto those transactions already committed in s', i.e., the graph $G(\text{CP}(s'))$, is acyclic. Thus, CP(s') $\in$ CSR. □

We next define *commit serializability* as a kind of "correction" for the situation discovered above for classes FSR and VSR, which intuitively requires that, in a correct execution, the already committed transactions have at any point in time been processed in a serializable way.

commit serializability

Definition 3.57 (Commit serializability)

A schedule s is *commit-serializable* if CP(s') is serializable for every prefix s' of s.

In detail, we need to distinguish the following classes of commit-serializable schedules:

CMFSR: class of all commit-final-state-serializable schedules
CMVSR: class of all commit-view-serializable schedules
CMCSR: class of all commit-conflict-serializable schedules

From this definition, we can immediately deduce the following statements:

Theorem 3.58

Let Full denote the set of all histories (i.e., complete schedules):

(i) Membership in any of the classes CMFSR, CMVSR, and CMCSR is prefix-commit-closed.

(ii) CMCSR $\subset$ CMVSR $\subset$ CMFSR

(iii) CMFSR $\cap$ Full $\subset$ FSR

(iv) CMVSR $\cap$ Full $\subset$ VSR

(v) CMCSR $\cap$ Full $=$ CSR

Proof. (i) and (ii) are straightforward from the definition of commit serializability as well as from the respective definitions of the underlying serializability notions and their relationships. The strictness of the inclusions mentioned in (ii) is demonstrated by the following sample histories:

$$s = w_1(x)r_2(x)w_2(y)w_1(y)c_1c_2w_3(x)w_3(y)c_3 \in \text{CMFSR} - \text{CMVSR}$$
$$s' = w_1(x)w_2(x)w_2(y)c_2w_1(y)w_3(x)w_3(y)c_3w_1(z)c_1 \in \text{CMVSR} - \text{CMCSR}$$

(iii) Let $s \in$ CMFSR $\cap$ Full. According to the definition of class CMFSR we then have CP$(s) \in$ FSR, since s is a prefix of itself. Let trans$(s) = \{t_1, \ldots, t_n\}$, and let w.l.o.g. commit$(s) = \{t_1, \ldots, t_k\}$, abort$(s) = \{t_{k+1}, \ldots, t_n\}$. Then CP$(s) \approx_f t_{\rho(1)} \ldots t_{\rho(k)}$ for a permutation ρ of $\{1, \ldots, k\}$, which implies

$$s \approx_f t_{\rho(1)} \ldots t_{\rho(k)} t_{k+1} \ldots t_n.$$

Therefore, $s \in$ FSR, or CMFSR $\cap$ Full $\subseteq$ FSR. The sample history s used above to prove that membership in FSR is not prefix-commit-closed suffices to show that this inclusion is even strict.

(iv) The proof is analogous to (iii).

(v) The proof of CMCSR $\cap$ Full $\subseteq$ CSR is also analogous to (iii). To show inclusion in the other direction consider some history $s \in$ CSR. Since membership in CSR is prefix-commit-closed, CP$(s') \in$ CSR for every prefix s' of s. Since s is by definition complete, the claim follows. □

The equality mentioned under (v) in the previous theorem has the following practical implication: Since commit-conflict-serializable histories are always conflict serializable and vice versa, conflict serializability is easily decided even for prefixes of a history by inspecting the projection onto committed transactions.

Wrapping up, we now have identified the relationships between classes of serializable and of commit-serializable histories as shown in Figure 3.13, where the histories mentioned are as follows:

summary of classes

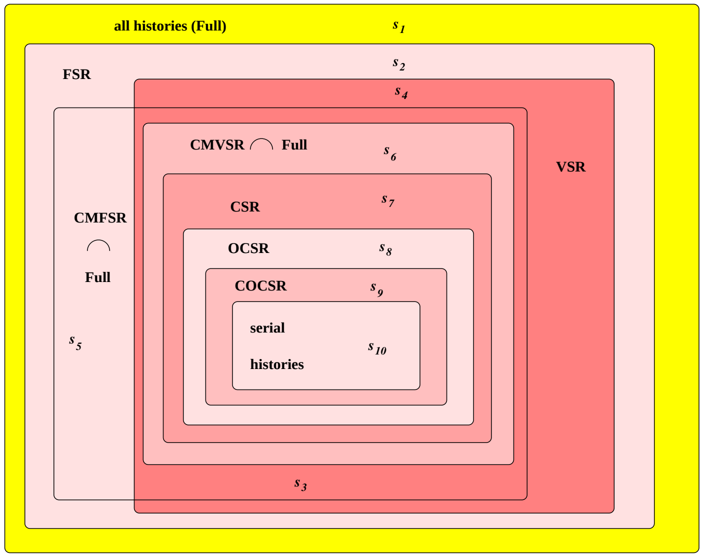

Figure 3.13: The landscape of serializable histories.

$s_1 = w_1(x)w_2(x)w_2(y)c_2w_1(y)c_1$
$s_2 = w_1(x)r_2(x)w_2(y)c_2r_1(y)w_1(y)c_1w_3(x)w_3(y)c_3$
$s_3 = w_1(x)r_2(x)w_2(y)w_1(y)c_1c_2$
$s_4 = w_1(x)w_2(x)w_2(y)c_2w_1(y)c_1w_3(x)w_3(y)c_3$
$s_5 = w_1(x)r_2(x)w_2(y)w_1(y)c_1c_2w_3(x)w_3(y)c_3$
$s_6 = w_1(x)w_2(x)w_2(y)c_2w_1(y)w_3(x)w_3(y)c_3w_1(z)c_1$
$s_7 = w_1(x)w_2(x)w_2(y)c_2w_1(z)c_1$
$s_8 = w_3(y)c_3w_1(x)r_2(x)c_2w_1(y)c_1$
$s_9 = w_3(y)c_3w_1(x)r_2(x)w_1(y)c_1c_2$
$s_{10} = w_1(x)w_1(y)c_1w_2(x)w_2(y)c_2$

3.10 An Alternative Correctness Criterion: Interleaving Specifications

We conclude this chapter by presenting another notion of correctness for transactions in the read-write model, which on one hand is less restrictive than conflict-based serializability, and on the other represents an attempt to take semantic information about transactions into account. Clearly, the possibilities to incorporate or exploit semantic information in the context of the read-write model are limited, but the considerations that follow can be, and have been, used in more general contexts as well.

Serializability is essentially based on the premise that a transaction has the ACID properties and in particular is a unit of isolation. In a correct schedule *all* steps of a transaction appear to be executed indivisibly to *all* other transactions in that schedule. There are two immediate generalizations to this: (i) consider units of isolation which do not necessarily consist of *all* the steps of a transaction, but of certain portions of the steps, and (ii) consider *varying* units of isolation w.r.t. other transactions, i.e., some steps may constitute an indivisible unit to some transactions, but not to others.

motivation

To motivate the discussion by way of a concrete application scenario, consider a bank where checking accounts belong to individuals, but savings accounts are shared by all members of a family. For simplicity, assume that the bank has only one couple as customers; so we need to manage two individual checking accounts, say a and b, and a common savings account, say c. Now consider three types of transactions on such accounts:

- A *transfer* transaction moves money from one of the checking accounts to the savings account. So, we obtain the following two transaction instances:
  $t_1 = r_1(a)w_1(a)r_1(c)w_1(c)$ and
  $t_2 = r_2(b)w_2(b)r_2(c)w_2(c)$.
- A *balance* transaction retrieves the balance of all accounts of the family, in our example:
  $t_3 = r_3(a)r_3(b)r_3(c)$.
- An *audit* transaction is initiated by the bank itself; it retrieves the balance of all accounts and writes the total into a special record, denoted z, for auditing purposes:
  $t_4 = r_4(a)r_4(b)r_4(c)w_4(z)$.

Now the question is whether this specific application bears some exploitable semantics which can be used when scheduling bank transactions. To this end, first notice

that a bank audit should have strong interleaving constraints; it should indeed be executed as an indivisible unit w.r.t. transfer transactions, and vice versa. However, the interleaving constraints for transfer and balance transactions may be less stringent, since family members are assumed to trust each other. For example, it could be tolerated that a balance transaction "misses" some money because of a concurrently ongoing transfer transaction. After all, a customer should be used to taking into account the possibility of money transfers by her or his spouse. With this intuition in mind, two sample schedules that the application could consider as acceptable are:

1. $r_1(a)w_1(a)r_2(b)w_2(b)r_2(c)w_2(c)r_1(c)w_1(c)$
2. $r_1(a)w_1(a)r_3(a)r_3(b)r_3(c)r_1(c)w_1(c)$

The first schedule is characteristic for the allowed interleavings of different transfer transactions. Note that this schedule is still conflict serializable. The second schedule, on the other hand, shows that a balance transaction can be interleaved with a transfer transaction in a relaxed manner such that the resulting execution is no longer conflict serializable, yet tolerable by the application. Of course, this shall not mean that all possible interleavings of these transactions are acceptable. For example, the following two example schedules should be disallowed:

1. $r_1(a)w_1(a)r_2(b)w_2(b)r_1(c)r_2(c)w_2(c)w_1(c)$
2. $r_1(a)w_1(a)r_4(a)r_4(b)r_4(c)w_4(z)r_1(c)w_1(c)$

The first of these two schedules would result in a lost-update anomaly, and the second one would not meet the stringent consistency demands of the bank's audit transaction.

disclaimer about immediate practicality

Note that in reality, the transactions would be specified as high-level programs, typically using the SQL language embedded in some host language (e.g., Java), and these programs would be parameterized. So the specific accounts that are involved in a transaction *instance* are merely the actual parameter values of a general pattern, and each of the above transaction *types* could spawn a large number of different transaction instances. In full generality, it could even be the case that the transactions that result from a single program differ not only in the arguments of their read and write actions, but also in their length and overall read/write pattern if the program contains conditional branches or loops. Exploiting the application-specific interleaving semantics would actually have to consider all possible transaction instances that result from a high-level program. In the following, however, we will disregard this substantial complication and rather assume that all transaction instances are explicitly given in advance. Indeed, we do not claim that the approach that we are going to develop in this section can be *directly* applied in practice. Nevertheless, the discussion will provide valuable insight into alternative correctness criteria.

formalization

In the sequel, we will formalize the approach by specifying for each pair of given transactions the indivisible units of one *relative to the other*. We can think of the borderline between two consecutive indivisible units of the same transaction as a "breakpoint" at which a transaction may be interleaved with another, and we can moreover think of such breakpoints as being specified *in advance*. In other words, if an application programmer who has to specify a transaction has sufficient knowledge of the semantics of the application for which the transaction is written, he or she may be able to state which interleavings with other transactions are acceptable, and which are not.

For ease of exposition, we consider a static situation in what follows in which a fixed set of transactions is given or known in advance. In addition, we restrict the attention to sequences of steps (i.e., total orders) when considering transactions, although the theory could also be generalized to partial orders.

indivisible units

Definition 3.59 (Indivisible units)

Let $T = \{t_1, \ldots, t_n\}$ be a set of transactions. For $t_i, t_j \in T$, $i \neq j$, an *indivisible unit of* t_i *relative to* t_j is a sequence of consecutive steps of t_i such that no operations of t_j are allowed to be executed within this sequence.
Let $\mathrm{IU}(t_i,\ t_j)$ denote the ordered sequence of indivisible units of t_i relative to t_j, and let $\mathrm{IU}_k(t_i,\ t_j)$ denote the k-th element of $\mathrm{IU}(t_i,\ t_j)$.

Example 3.60

Consider the following three transactions:

$$t_1 = r_1(x)w_1(x)w_1(z)r_1(y)$$
$$t_2 = r_2(y)w_2(y)r_2(x)$$
$$t_3 = w_3(x)w_3(y)w_3(z)$$

The following specification may be given for these transactions relative to each other:

$$\mathrm{IU}(t_1,\ t_2) = \langle\ [r_1(x)w_1(x)],\ [w_1(z)r_1(y)]\ \rangle$$
$$\mathrm{IU}(t_1,\ t_3) = \langle\ [r_1(x)w_1(x)],\ [w_1(z)],\ [r_1(y)]\ \rangle$$

$$\mathrm{IU}(t_2,\ t_1) = \langle\ [r_2(y)],\ [w_2(y)r_2(x)]\ \rangle$$
$$\mathrm{IU}(t_2,\ t_3) = \langle\ [r_2(y)w_2(y)],\ [r_2(x)]\ \rangle$$

$$\mathrm{IU}(t_3,\ t_1) = \langle\ [w_3(x)w_3(y)],\ [w_3(z)]\ \rangle$$
$$\mathrm{IU}(t_3,\ t_2) = \langle\ [w_3(x)w_3(y)],\ [w_3(z)]\ \rangle$$

For example, $\mathrm{IU}(t_1,\ t_2)$ is meant to say that if operations from t_2 have to be executed within t_1, then they may only be executed between $w_1(x)$ and $w_1(z)$, i.e., between $\mathrm{IU}_1(t_1,\ t_2) = [r_1(x)w_1(x)]$ and $\mathrm{IU}_2(t_1,\ t_2) = [w_1(z)r_1(y)]$. □

We will say that an operation $q \in t_j$ is *interleaved with* the element $\mathrm{IU}_k(t_i,\ t_j)$ of $\mathrm{IU}(t_i,\ t_j)$ in the context of a schedule s if there exist operations p and p' from $\mathrm{IU}_k(t_i,\ t_j)$ s.t. $p <_s q$ and $q <_s p'$.

Clearly, the meaning of a given indivisible unit $\mathrm{IU}_k(t_i,\ t_j)$ will be that, for a schedule s to qualify for being "correct," no operations from t_j are interleaved in s with that unit.

Example 3.61

Consider $\mathrm{IU}(t_1,\ t_2)$ in the Example 3.60: In a correct schedule, no operation from t_2 may occur inside any of the two units $\mathrm{IU}_1(t_1,\ t_2)$ and $\mathrm{IU}_2(t_1,\ t_2)$, but only before, after, or in between. Indeed, consider the schedule

$$s_1 = r_2(y)r_1(x)w_1(x)w_2(y)r_2(x)w_1(z)w_3(x)w_3(y)r_1(y)w_3(z).$$

Apparently, s_1 is not a serial schedule and not even a serializable one; however, it is correct w.r.t. the relative isolation specification given in the previous example: Operations of t_1 are executed between $r_2(y)$ and $w_2(y)r_2(x)$, i.e., the isolation of t_2 relative to t_1 follows the specification and is hence preserved. As is easily seen, the same holds for the other interleavings in this schedule.
As a second example, consider the schedule

$$s_2 = r_1(x)r_2(y)w_2(y)w_1(x)r_2(x)w_1(z)r_1(y)$$

that interleaves the two transactions t_1 and t_2 in a way that would not be compliant with the allowed interleavings according to $\mathrm{IU}(t_1, t_2)$. Note, however, that this schedule could be easily transformed into another schedule s_2' by exchanging $w_1(x)$ with all actions of t_2 such that $w_1(x)$ would precede t_2. This transformation seems perfectly legal as all actions of t_2 are conflict-free (i.e., commutative) with regard to $w_1(x)$, and the resulting schedule s_2' would be acceptable according to $\mathrm{IU}(t_1, t_2)$. □

The example of schedule s_2 above shows that it is desirable to combine the "traditional" reasoning about conflicts with the specification of indivisible units in order to allow more concurrent executions. In the following, we are aiming at a notion of serializability that is based on an equivalence relation to serial executions and reconciles interleaving specifications with conflict equivalence.

depends-on relation on steps

Definition 3.62 (Dependence of steps)

Let s be schedule.

(i) A step q *directly depends on* a step p in s, denoted $p \rightsquigarrow q$, if $p <_s q$ and either $p, q \in t$ for some transaction $t \in \text{trans}(s)$ and $p <_t q$, or p and q are from distinct transactions and p is in conflict with q.

(ii) Let $\overset{*}{\rightsquigarrow}$ ("*depends on*") denote the reflexive and transitive closure of $\rightsquigarrow$.

Thus, the *depends on* relation of a schedule is derived from conflicts and the internal structure of transactions.

The depends relation between steps now allows us to define the analogon to serial schedules for the setting considered here. In a "relatively serial" schedule, an operation q from a transaction t_j may be interleaved with an indivisible unit of a transaction t_i relative to t_j if q is neither depending on an operation in that unit, nor does an operation from that unit depend on q.

relatively serial schedule

Definition 3.63 (Relatively serial schedule)

A schedule s s.t. $\text{trans}(s) = T$ is *relatively serial* if for all transactions $t_i, t_j \in T$, $i \neq j$, the following holds: If an operation $q \in t_j$ is interleaved with $\text{IU}_k(t_i,\, t_j)$ for some k, then there is no operation $p \in \text{IU}_k(t_i,\, t_j)$ $(p \in t_i)$ s.t. $p \overset{*}{\rightsquigarrow} q$ or $q \overset{*}{\rightsquigarrow} p$.

Example 3.64

Consider the specification from Example 3.60 again as well as the following schedule:

$$s_3 = r_1(x)r_2(y)w_1(x)w_2(y)w_3(x)w_1(z)w_3(y)r_2(x)r_1(y)w_3(z)$$

Here operation $r_2(y)$ is interleaved with $\text{IU}_1(t_1,\, t_2) = [r_1(x)w_1(x)]$, but notice that neither $r_1(x) \overset{*}{\rightsquigarrow} r_2(y)$ nor $r_2(y) \overset{*}{\rightsquigarrow} w_1(x)$ holds. Similarly, $w_1(z)$ is interleaved with $\text{IU}_2(t_2,\, t_1) = [w_2(y)r_2(x)]$ and $\text{IU}_1(t_3,\, t_1) = [w_3(x)w_3(y)]$, but there are no dependencies between the operations involved. Finally, $r_2(x)$ is interleaved with $\text{IU}_2(t_1,\, t_2) =$

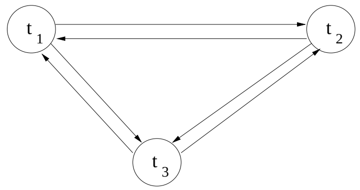

Figure 3.14: Conflict graph of schedule s_3 from Example 3.64.

$[w_1(z)r_1(y)]$, but again neither $w_1(z) \overset{*}{\rightsquigarrow} r_2(x)$ nor $r_2(x) \overset{*}{\rightsquigarrow} r_1(y)$ holds. Thus, schedule s_3 is relatively serial according to Definition 3.63. On the other hand, the conflict graph of s_3 is shown in Figure 3.14. The conflict graph is obviously cyclic, so that $s_3 \notin \text{CSR}$. □

We should think of a relatively serial schedule as the analog of a serial schedule in the theory we have developed earlier in this chapter. Thus, schedules which are relatively serial should always be acceptable, and schedules which are not relatively serial should be acceptable as long as they are conflict equivalent to a relatively serial schedule:

relative serializability

Definition 3.65 (Relative serializability)

A schedule s is *relatively serializable* if it is conflict equivalent to some relatively serial schedule.

Example 3.66

Consider the following schedule over the specification from Example 3.60:

$$s_4 = r_1(x)r_2(y)w_2(y)w_1(x)w_3(x)r_2(x)w_1(z)w_3(y)r_1(y)w_3(z)$$

Notice that s_4 is no longer relatively serial, since now $w_1(x)$ is interleaved with $\text{IU}_2(t_2,\, t_1) = [w_2(y)r_2(x)]$, and $w_1(x) \overset{*}{\rightsquigarrow} r_2(x)$. However, s_4 is conflict equivalent to schedule s_3 from Example 3.64, since steps $w_2(y)$ and $r_2(x)$ have only been moved beyond other steps with which they commute. Thus, s_4 is relatively serializable. □

We mention that the classical theory of conflict serializable schedules as developed in Section 3.8 is a special case of what we have decribed in this section. Indeed, traditionally entire transactions are the only units of isolation, so that every serial schedule is also relatively serial since each operation of a serial schedule is not interleaved with the single indivisible unit of any transaction. However, it is easy to see that not every relatively serial schedule is serial under traditional isolation, basically because relatively serial schedules still allow interleavings of operations with the indivisible unit of other transactions that do not have any dependencies between them.

If a given schedule is not relatively serial, it may still be relatively serializable and hence acceptable from a semantic point of view. The latter may be the case if there are interleavings of operations with indivisible units of other transactions that should be forbidden (or at least avoided) due to dependencies between the operations involved. In particular, an operation which is interleaved with an indivisible unit can sometimes be "moved out" of the indivisible unit by allowed commutations.

Example 3.67

Consider an indivisible unit $[w_1(x)r_1(z)]$ of a transaction t_1 w.r.t. a transaction t_2, and let

$$w_1(x) <_s r_2(x) <_s r_1(z)$$

occur in a schedule s. Clearly, that schedule cannot be relatively serial since $w_1(x) \overset{*}{\rightsquigarrow} r_2(x)$, but if the context of the schedule does not require otherwise, we can commute $r_2(x)$ and $r_1(z)$ in order to obtain

$$[w_1(x)r_1(z)] <_s r_2(x).$$

So, in a sense, we have pushed $r_2(x)$ forward in a conflict-preserving way and thereby (locally) restored relative seriality.

push forward

Next suppose the same schedule s also contains a situation

$$r_3(z) <_s w_2(y) <_s r_3(y),$$

but the specification requires that $[r_3(z)r_3(y)]$ is an indivisible unit w.r.t. t_2. By a similar argument, we can correct this situation by pulling $w_2(y)$ backward in front of the indivisible unit, again by an allowed commutation. We obtain

pull backward

$$w_2(y) <_s [r_3(z)r_3(y)],$$

which again contributes to making this particular schedule acceptable. □

We now formalize these observations by introducing the notions of pushing *forward* or pulling *backward* an operation. We will then be able to introduce a graph which contains edges that represent these actions, and that can be used to characterize relative serializability.

Definition 3.68 (Push forward and pull backward)

Let t_i and t_j be distinct transactions, and let $\mathrm{IU}_k(t_i, t_j)$ be an indivisible unit of t_i relative to t_j. For an operation $p_i \in \mathrm{IU}_k(t_i, t_j)$ $(p_i \in t_i)$ let

(i) $F(p_i, t_j)$ be the last operation in $\mathrm{IU}_k(t_i, t_j)$,

(ii) $B(p_i, t_j)$ be the first operation of $\mathrm{IU}_k(t_i, t_j)$.

Intuitively, $F(p_i, t_j)$ denotes the earliest operation from t_i beyond which an operation from t_j interleaved with $\mathrm{IU}_k(t_i, t_j)$ can be pushed forward in order to restore relative isolation, and correspondingly $B(p_i, t_j)$ denotes the latest operation from t_i before which an operation from t_j interleaved with $\mathrm{IU}_k(t_i, t_j)$ can be pulled backward. In Example 3.60 we have for $\mathrm{IU}(t_1, t_2)$

$$F(r_1(x), t_2) = w_1(x) \quad \text{and}$$
$$B(r_1(y), t_2) = w_1(z).$$

We will now use these notions for capturing graphically ways to reorganize a given schedule so as to make it relatively serial, or to test a schedule for relative serializability:

relative serialization graph

Definition 3.69 (Relative serialization graph)

Let s be a schedule. The *relative serialization graph* $RSG(s) = (V, E)$ of s is defined by $V := \mathrm{op}(s)$ and $E \subseteq V \times V$ containing four types of edges as follows:

(i) If p and q are consecutive operations of the same transaction in s, then $(p, q) \in E$ (internal or I-edge);

(ii) if $p \overset{*}{\rightsquigarrow} q$ for $p \in t_i$, $q \in t_j$, $i \neq j$, then $(p, q) \in E$ (dependency or D-edge);

(iii) if (p, q) is a D-edge s.t. $p \in t_i$ and $q \in t_j$, then $(F(p, t_j), q) \in E$ (push-forward or F-edge);

(iv) if (p, q) is a D-edge s.t. $p \in t_i$ and $q \in t_j$, then $(p, B(q, t_i)) \in E$ (pull-backward or B-edge).

Example 3.70

Consider the following three transactions:

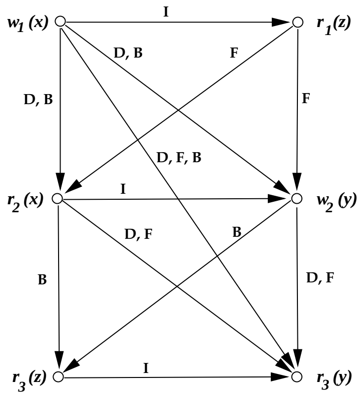

Figure 3.15: Relative serialization graph of schedule s_5 from Example 3.70.

$$t_1 = w_1(x)r_1(z)$$
$$t_2 = r_2(x)w_2(y)$$
$$t_3 = r_3(z)r_3(y)$$

Let the following interleaving specification be given for these transactions:

$$\mathrm{IU}(t_1, t_2) = \langle\ [w_1(x), r_1(z)]\ \rangle$$
$$\mathrm{IU}(t_1, t_3) = \langle\ [w_1(x)],\ [r_1(z)]\ \rangle$$

$$\mathrm{IU}(t_2, t_1) = \langle\ [r_2(x)],\ [w_2(y)]\ \rangle$$
$$\mathrm{IU}(t_2, t_3) = \langle\ [r_2(x)],\ [w_2(y)]\ \rangle$$

$$\mathrm{IU}(t_3, t_1) = \langle\ [r_3(z)],\ [r_3(y)]\ \rangle$$
$$\mathrm{IU}(t_3, t_2) = \langle\ [r_3(z)r_3(y)]\ \rangle$$

Now consider the following schedule:

$$s_5 = w_1(x)r_2(x)r_3(z)w_2(y)r_3(y)r_1(z)$$

Notice that we have analyzed the situation we see in this schedule in part in Example 3.67 already. The relative serialization graph of s_5 is shown in Figure 3.15. □

Lemma 3.71

If s is relatively serial, then $RSG(s)$ is acyclic.

Proof. We show for $RSG(s) = (V,\ E)$ that

$$(p, q) \in E \text{ implies } p <_s q.$$

This is trivial for I- as well as for D-edges. Hence consider an F-edge of the form (r, q), where $q \in t_j$, $r = F(p, t_j)$, $r, p \in t_i$, and (p, q) is a D-edge. This situation implies that $p, r \in \mathrm{IU}_k(t_i, t_j)$, $p <_s r$, and r is the last operation of this unit. Since $p \overset{*}{\rightsquigarrow} q$ and s is relatively serial, q appears after $\mathrm{IU}_k(t_i, t_j)$ in s. Thus, $r <_s q$. The corresponding statement for B-edges is shown similarly. □

The following can now be proven:

Theorem 3.72

A schedule s is relatively serializable iff $RSG(s)$ is acyclic.

Proof. (only if) If s is relatively serializable, then there exists an s' which is relatively serial s.t. s and s' are conflict equivalent. We first show $RSG(s) = RSG(s')$. To this end, both graphs have the same set of nodes, and both have the same I-edges. Morevoer, D-edges are the same in both graphs due to equal I-edges as well as to conflict equivalence of the two schedules. Since F- and B-edges depend on the D-edges and on the indivisible units, the claim follows. Now since s' is relatively serial, $RSG(s')$ is acyclic by the previous lemma. Thus, $RSG(s)$ is acyclic as well.

(if) Let $RSG(s)$ be acyclic, and let s' be obtained by topologically sorting $RSG(s)$. Clearly, s and s' are conflict equivalent and hence $RSG(s) = RSG(s')$. It remains to be shown that s' is relatively serial. Assume, on the contrary, that it is not, i.e., there exists a unit $\mathrm{IU}_k(t_i, t_j)$ for some k, t_i, and t_j that has some $q \in t_j$ interleaved, but there is also some $p \in t_i$ in $\mathrm{IU}_k(t_i, t_j)$ s.t. $p \overset{*}{\rightsquigarrow} q$ or $q \overset{*}{\rightsquigarrow} p$ in s'. Let r be the last operation in $\mathrm{IU}_k(t_i, t_j)$. If $p \overset{*}{\rightsquigarrow} q$, then there is an F-edge (r, q) in $RSG(s)$; hence, in *any* topological sort of $RSG(s)$, we have $r < q$. However, in s', which is such a topological sort, we find $q < r$ due to the assumed interleaving, which gives a contradiction. A similar argument applies if $q \overset{*}{\rightsquigarrow} p$, since $RSG(s)$ in that case has a corresponding B-edge. The theorem follows. □

Since the relative serialization graph of a schedule can be constructed efficiently, the characterization given in Theorem 3.72 provides a test for relative serializability which is polynomial in op(s). As an example, schedule s_4 above, which has an acyclic relative serialization graph, is relatively serializable.

3.11 Lessons Learned

We conclude this chapter by summarizing the discoveries we have made. First of all, the page-level transaction model, which may seem very simple and insufficient at first glance, turns out to give rise to a rich theory of serializability that allows for numerous concepts and approaches to schedule correctness. However, parts of this theory (in particular those leading to classes FSR and VSR), although seemingly appropriate from a "semantic" point of view, are unrealistic for practical purpose, so that conflict serializability is the correctness notion of choice for the construction of schedulers. This is due to the high (and hence unacceptable) complexity of the decision problems related to final-state as well as to view serializability.

Conflict serializability has a number of interesting properties which makes it particularly amenable to real-world implementations: its decision problem is polynomial in the number of transactions considered, it enjoys desirable closure properties, and it allows establishing subclasses, or sufficient conditions, that capture additional and relevant properties. The ones we have seen here are order-preserving conflict serializability and commit-order preservation.

In addition, the previous section has also uncovered a first attempt to go beyond traditional conflict-based serializability theory. Indeed, the idea of taking semantic information of the database or of the transactions (or maybe even of the application programs from which the transactions are issued) into account when considering the correctness of schedules is nearly as old as the notion of serializability itself, and relative serializability constitutes an interesting approach in this direction. Clearly, as the read-write model of transactions is of a highly syntactic nature, more can be expected in terms of semantic considerations when higher-level transaction models are employed.

We will re-use the notion of conflict serializability several times in subsequent chapters. In particular, we will demonstrate for completely different types of operations, e.g., update operations along the lines of SQL for relational databases, that a theory of serializability can mostly follow what we have described in this chapter, and that "serializability theorems" in the style of Theorem 3.41 which characterize (conflict) serializability in graph-theoretic terms can be obtained for a variety of settings. In this sense, what we have done and seen in this chapter is on the one hand a major preparation for the practical aspects of concurrency control and of scheduler design we will study next, but is on the other hand representative for a kind of "methodology" that is available for deriving transaction models and their correctness criteria.

Exercises

1. Consider the following histories:

   $s = r_1(x)r_2(y)w_1(y)r_3(z)w_3(z)r_2(x)w_2(z)w_1(x)c_1c_2c_3$
   $s' = r_3(z)w_3(z)r_2(y)r_2(x)w_2(z)r_1(x)w_1(y)w_1(x)c_3c_2c_1$

   Compute $H[s]$ and $H[s']$ as well as the respective RF and LRF relations. Show the step graphs of these histories.

2. Suppose that the following schedule is given:

   $s = r_1(x)r_3(x)w_3(y)w_2(x)c_3r_4(y)w_4(x)c_2r_5(x)c_4w_5(z)w_1(z)c_1c_5$

   Now assume that it is known that for some transactions, the function computed by a write step is the *identity* function in one of its arguments, i.e., for $w_i(x)$ we have that $f_{ix}(v_1, \ldots, v_m) = v_j$ for some index j (such a function is a *copier* since it just copies information it has read into a data item). Compute the (Herbrand) semantics of s given the information that t_3 and t_4 are copiers. *Note:* Strictly speaking, this is no longer a Herbrand semantics, since some function symbols in $H[s]$ are now interpreted.

3. Suppose that in a given schedule the functions corresponding to the write steps represent increments of a counter, i.e., $f(x) = x + 1$. Compute the (Herbrand) semantics of the following schedules using this semantic information:

   $s = r_3(z)r_1(y)w_3(z)w_1(y)r_1(x)r_2(y)w_2(y)w_1(x)r_2(x)w_2(x)c_1c_2c_3$
   $s' = r_3(z)w_3(z)r_2(y)w_2(y)r_1(y)w_1(y)r_2(x)w_2(x)r_1(x)w_1(x)c_3c_2c_1$

4. Consider the following history:

   $s = r_1(x)r_3(x)w_3(y)w_2(x)r_4(y)c_2w_4(x)c_4r_5(x)c_3w_5(z)c_5w_1(z)c_1$

   Into which of the classes FSR, VSR, CSR does this schedule fall?

5. Complete the proof of Theorem 3.22, i.e., show that for any two schedules s and s', $s \approx_v s'$ iff $D(s) = D(s')$.

6. Consider $s = r_1(x)w_1(x)r_2(x)r_2(y)w_2(y)c_2w_1(y)c_1$. Show that $s \in$ FSR–VSR.

7. Prove Theorem 3.26, i.e., show that VSR and FSR coincide in the absence of dead steps.

8. Show that VSR = CSR in the absence of blind writes, i.e., if each write step on a data item x is preceded by a read step on x of the same transaction.

9. Let $s = r_1(z)r_3(x)r_2(z)w_1(z)w_1(y)c_1w_2(y)w_2(u)c_2w_3(y)c_3$.

Show $s \in$ VSR–CSR.

10. Consider $s = r_1(x)w_1(z)r_2(z)w_1(y)c_1r_3(y)w_2(z)c_2w_3(x)w_3(y)c_3$. Using the conflict graph of s as an argument, show that $s \in$ CSR. Does $s \in$ OCSR also hold?

11. Show:

    (i) Membership in class CSR is monotone.

    (ii) $s \in \text{CSR} \iff (\forall\, T \subseteq \text{trans}(s))\ \Pi_T(s) \in \text{VSR}$
    (i.e., CSR is the largest monotone subset of VSR).

12. Consider the traditional case where entire transactions are the only units of indivisibility w.r.t. other transactions. Show by way of an example that, in this setting, not every relatively serial schedule is also serial. Also show that under this kind of "absolute" isolation, every relatively serial schedule is conflict equivalent to some serial schedule.

Bibliographical and Other Remarks

Our exposition of the Herbrand semantics of schedules as well as that of the classes FSR, VSR, and CSR mostly follows Papadimitriou (1986) as well as Hadzilacos (1988). A major difference between the two is that the former ignores aborted transactions in the discussion of serializability, while the latter does not. The subtle difference between these viewpoints already comes up in the definition of reading a value from a write step, which has to be written as follows when aborted transactions are taken into account:

> A step $r_i(x) \in s$ of a transaction $t_i \in \text{trans}(s)$ reads the value written by the *last* $w_j(x) \in s, j \neq i$, that occurs before $r_i(x)$, such that $a_j \notin s$,

The seemingly innocent addition at the end that t_j should not be aborted in schedule s has interesting consequences. Indeed, consider the following (fragment of a) schedule:

$$s_1 = w_i(x)w_j(x)r_k(x)$$

According to what was said above, step $r_k(x)$ reads the value of x written by $w_j(x)$, since (i) there is no other step in between the two, and (ii) a_j is not around. Next suppose a scheduler appends a_j to s_1 to obtain

$$s_2 = w_i(x)c_iw_j(x)r_k(x)a_j.$$

Now since t_j is aborted, step $r_k(x)$ no longer reads from $w_j(x)$, but from (the even committed) step $w_i(x)$. If a_j would have been there just a little earlier, as in

$$s_3 = w_i(x)c_iw_j(x)a_jr_k(x),$$

it would have been clear from the outset that $r_k(x)$ does not read from $w_j(x)$. The problem we encounter in this little discussion is that the "reads-from definition" above formally introduces a well-defined notion (since $s_1 \neq s_2$); however, in a dynamic situation s_2 could be produced from s_1, and then the value read by $r_k(x)$ changes! We take this as evidence that failures are difficult to model, at least in a setting (as discussed in this chapter) where correctness of transaction executions is the focus, and have therefore decided not to consider aborted transactions in this chapter.

Presumably the first publication that used the Herbrand semantics to characterize the correctness of schedules was by Casanova and Bernstein (1980). The notion of (final-state) serializability along with the relations RF and LRF has been introduced by Papadimitriou (1979) as well as Bernstein, Shipman, and Wong (1979). The notion of view serializability has been made fully explicit by Yannakakis (1984). Vidyasankar (1987) extends this work as well as that of Ibaraki et al. (1983) and Katoh et al. (1985) by introducing a generalized notion of *S-serializability* where the subset S of the set of transactions under consideration reads the same values in a given schedule as in some serial execution. S-serializability is then characterized in terms of an acyclic *transaction precedence graph* that expresses useful reads as well as useless writes. Moreover, S-serializability coincides with view serializability if S contains all transactions in question.

The step-graph construction and the proof that equality of reduced step graphs is equivalent to the fact that the histories in question are final-state serializable is due to Papadimitriou (1986). The NP-completeness result on view serializability goes back to Papadimitriou (1979); a complete proof appears in Papadimitriou (1986). The latter source also proves that conflict equivalence can be characterized in terms of commutativity. The related notion of weak view serializability of degree k (introduced in the Exercises above) is from Tuzhilin and Spirakis (1985). The practically most relevant notion of conflict serializability and the characterization of membership in CSR by the acyclicity of the associated conflict graph goes back to the seminal paper by Eswaran et al. (1976).

Order-preserving serializability was already discussed by Papadimitriou (1979) as well as Bernstein, Shipman, and Wong (1979) under the name of "strict" serializability; we should mention that order preservation not only makes sense for conflict serializable schedules, but also for those in FSR or in VSR, which then gives rise to restricted classes OFSR and OVSR, resp. While the complexity of testing membership in OFSR or OVSR remains open in Papadimitriou (1979), Sethi (1982) clarifies this by showing that order-preserving serializability of a history s can be decided in polynomial time for transactions in the two-step model (see Exercises in Chapter 2), provided all write steps in s are alive; the problem is NP-complete in general.

The property of commit-order preservation, discussed in Section 3.8.4.2, was proposed by Weihl (1989), Breitbart and Silberschatz (1991) as well as Raz (1992, 1994) under different names; Raz (1992, 1994) and Guerraoui (1995) also investigate a number of implications of this notion.

Our discussion of commit serializability again follows Hadzilacos (1988). A proof of the fact that CSR is the largest monotone subset of VSR (see Exercise 11) can be found in Yannakakis (1984).

Our presentation in Section 3.10 mostly follows Agrawal et al. (1994), Krishnaswamy et al. (1997) as well as Vidyasankar (1998). The idea of letting the user specify which interleavings of transactions are acceptable or of deriving such knowledge from the underlying application was first discussed by Garcia-Molina (1983) and then studied in detail by Lynch (1983). Lynch (1983) in particular takes the idea of relative serializability across multiple levels of abstraction and consequently establishes a notion of "multi-level atomicity". This theory, which we have here restricted to the (lowest) read-write level only, is also discussed by Cellary et al. (1988). Relative serializability was first established by Farrag and Özsu (1989), who use the notion of *breakpoints* for separating atomic units of a transaction t_i relative to another transaction t_j. Agrawal et al. (1994) extend this by allowing non-dependent operations to interleave anywhere, and by basing dependency between operations on precedence as well as on conflict. Vidyasankar (1998) defines breakpoints relative to individual operations, not entire transactions, which may result in multiple F- and B-edges between nodes in a relative serialization graph.