Contents

Index

2 SERIALIZABILITY THEORY

2.1 HISTORIES

Serializability theory is a mathematical tool that allows us to prove whether or not a scheduler works correctly. In the theory, we represent a concurrent execution of a set of transactions by a structure called a history. A history is called serializable if it represents a serializable execution. The theory gives precise properties that a history must satisfy to be serializable.

Transactions

We begin our development of serializability theory by describing how transactions are modelled. As we said in Chapter 1, a transaction is a particular execution of a program that manipulates the database by means of Read and Write operations. From the viewpoint of serializability theory, a transaction is a representation of such an execution that identifies the Read and Write operations and indicates the order in which these operations execute. For each Read and Write, the transaction specifies the name, but not the value, of the data item read and written (respectively). In addition, the transaction contains a Commit or Abort as its last operation, to indicate whether the execution it represents terminated successfully or not.

For example, an execution of the following program

```
Procedure P begin
    Start;
    temp := Read(x);
    temp := temp + 1;
    Write(x, temp);
    Commit
end
```

may be represented as: $r_1[x] \rightarrow w_1[x] \rightarrow c_1$. The subscripts identify this particular transaction and distinguish it from other transactions that happen to access the same data items in the same order—for instance, other executions of the same program.

In general, we use $r_i[x]$ (or $w_i[x]$) to denote the execution of a Read (or Write) issued by transaction T_i on data item x. To keep this notation unambiguous, we assume that no transaction reads or writes a data item more than once. None of the results in this chapter depend on this assumption (see Exercise 2.10). We use c_i and a_i to denote T_i's Commit and Abort operations (respectively). In a particular transaction, only one of these two can appear. The arrows indicate the order in which operations execute. Thus in the example, $w_1[x]$ follows ("happens after") $r_1[x]$ and precedes ("happens before") c_1.

As we saw in Chapter 1, a transaction may be generated by concurrently executing programs. For example, a program that reads data items x and y and writes their sum into z might issue the two Reads in parallel. This type of execution is modelled as a *partial* order. In other words, the transaction need not specify the order of every two operations that appear in it. For instance, the transaction just mentioned would be represented as:

$$\begin{tikzcd} r_2[x] \arrow[dr] & & \\ & w_2[z] \arrow[r] & c_2 \\ r_2[y] \arrow[ur] & & \end{tikzcd}$$

This says that $w_2[z]$ must happen after both $r_2[x]$ and $r_2[y]$, but that the order in which $r_2[x]$ and $r_2[y]$ take place is unspecified and therefore arbitrary.

If a transaction both reads and writes a data item x, we require that the partial order specify the relative order of Read(x) and Write(x). This is because the order of execution of these operations necessarily matters. The value of x returned by Read(x) depends on whether this operation precedes or follows Write(x).

We want to formalize the definition of a transaction as a partial ordering of operations. In mathematics, it is common practice to write a partial order as an ordered pair $(\Sigma, <)$, where Σ is the set of elements being ordered and $<$ is the ordering relation.[1] In this notation, we would define a transaction T_i to be an ordered pair $(\Sigma_i, <_i)$, where Σ_i is the set of operations of T_i and $<_i$ indicates the execution order of those operations.

This notation is a bit more complex than we need. We can do away with the symbol Σ by using the name of the partial order, in this case T_i, to denote *both* the partial order and the set of elements (i.e., operations) in the partial order. The meaning of a symbol that denotes both a partial order and its elements, such as T_i, will always be clear from context. In particular, when we write $r_i[x] \in T_i$, meaning that $r_i[x]$ is an element (i.e., operation) of T_i, we are

[1] The definition of partial orders is given in Section A.4 of the Appendix.

using T_i to denote the set of operations in the partial order. We are now ready to give the formal definition.

A *transaction* T_i is a partial order with ordering relation $<_i$ where

1. $T_i \subseteq \{r_i[x], w_i[x] \mid x \text{ is a data item}\} \cup \{a_i, c_i\}$;
2. $a_i \in T_i$ iff $c_i \notin T_i$;
3. if t is c_i or a_i (whichever is in T_i), for any other operation $p \in T_i$, $p <_i t$; and
4. if $r_i[x], w_i[x] \in T_i$, then either $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$.

In words, condition (1) defines the kinds of operations in the transaction. Condition (2) says that this set contains a Commit or an Abort, but not both. $<_i$ indicates the order of operations. Condition (3) says that the Commit or Abort (whichever is present) must follow all other operations. Condition (4) requires that $<_i$ specify the order of execution of Read and Write operations on a common data item.

We'll usually draw transactions as in the examples we've seen so far, that is, as directed acyclic graphs[2] (dags) with the arrows indicating the ordering defined by $<_i$. To see the relationship between the two notations, consider the following transaction.

$$\begin{tikzcd}
r_2[x] \arrow[dr] & & \\
& w_2[z] \arrow[r] & c_2 \\
r_2[y] \arrow[ur] & &
\end{tikzcd}$$

Formally, this says that T_2 consists of the operations $\{r_2[x], r_2[y], w_2[z], c_2\}$ and $<_2 = \{(r_2[x], w_2[z]), (r_2[y], w_2[z]), (w_2[z], c_2), (r_2[x], c_2), (r_2[y], c_2)\}$.[3] Note that we generally do not draw arcs implied by transitivity. For example, the arc $r_2[x] \rightarrow c_2$ is implied by $r_2[x] \rightarrow w_2[z]$ and $w_2[z] \rightarrow c_2$.

Our formal definition of a transaction does not capture every observable aspect of the transaction execution it models. For example, it does not describe the initial values of data items or the values written by Writes. Moreover, it only describes the database operations, and not, for example, assignment or conditional statements. Features of the execution that are not modelled by transactions are called *uninterpreted*, meaning unspecified. When analyzing transactions or building schedulers, we must be careful not to make assumptions about uninterpreted features. For example, we must ensure that our analysis holds for all possible initial states of the database and for all possible computations that a program might perform in between issuing its Reads and Writes. Otherwise, our analysis may be incorrect for some database states or computations.

[2] The definition of dags and their relationship to partial orders is given in Sections A.3 and A.4 of the Appendix.

[3] A standard notation for a binary relation $<_2$ is the set of pairs (x, y) such that $x <_2 y$.

For example, in the transaction $r_i[x] \rightarrow w_i[x] \rightarrow c_i$, we cannot make any assumptions about the initial value of x or the computation performed by T_i in between the Read and Write. In particular, we cannot tell whether or not the value written into x by the Write depends on the value of x that was read. To ensure that our analysis is valid for all interpretations of a transaction, we assume that each value written by a transaction depends on the values of all the data items that it previously read. To put it more formally, for every transaction T_i and for all data items x and y, the value written by $w_i[x]$ is an arbitrary function of the values read by all $r_i[y] <_i w_i[x]$.

You may object that the value written by $w_i[x]$ might also depend on information supplied to T_i by input statements executed before $w_i[x]$. This is true and is a good reason for including input statements in transactions. But we can take care of this without expanding the repertoire of operations by modelling input statements as Reads and output statements as Writes. Each such Read or Write operates on a *unique* data item, one that is referenced by no other operation. These data items must be unique to accurately model the fact that a value read from or written to a terminal or similar I/O device by one transaction isn't read or written by any other transaction. For example, a Write to a terminal produces a value that is not read by any subsequent Read on that terminal. In this way we can incorporate input and output statements in our model of executions without complicating the model.

The choice of what information to incorporate in a formal model of a transaction and what information to leave out is based on the scheduler's view of the system. Our model includes only those aspects of a transaction that we choose to allow the DBS's scheduler to exploit when trying to attain a serializable execution. Of course, we must give the scheduler enough information to successfully avoid nonserializable executions. As we will see, we have defined transactions in a way that satisfies this requirement.

Histories

When a set of transactions execute concurrently, their operations may be interleaved. We model such an execution by a structure called a history. A history indicates the order in which the operations of the transactions were executed relative to each other. Since some of these operations may be executed in parallel, a history is defined as a *partial* order. If transaction T_i specifies the order of two of its operations, these two operations must appear in that order in any history that includes T_i. In addition, we require that a history specify the order of all *conflicting operations* that appear in it.

Two operations are said to *conflict* if they both operate on the same data item and at least one of them is a Write. Thus, Read(x) conflicts with Write(x), while Write(x) conflicts with both Read(x) and Write(x). If two operations conflict, their order of execution matters. The value of x returned by Read(x) depends on whether or not that operation precedes or follows a particular

Write(x). Also, the final value of x depends on which of two Write(x) operations is processed last.

Formally, let $T = \{T_1, T_2, ..., T_n\}$ be a set of transactions. A *complete history* H over T is a partial order with ordering relation $<_H$ where:

1. $H = \cup_{i=1}^{n} T_i$;
2. $<_H \supseteq \cup_{i=1}^{n} <_i$; and
3. for any two conflicting operations $p, q \in H$, either $p <_H q$ or $q <_H p$.

Condition (1) says that the execution represented by H involves precisely the operations submitted by $T_1, T_2, ..., T_n$. Condition (2) says that the execution honors all operation orderings specified within each transaction. Finally, condition (3) says that the ordering of every pair of conflicting operations is determined by $<_H$. When the history under consideration is clear from the context, we drop the H subscript from $<_H$.

A *history* is simply a prefix of a complete history.[4] Thus a history represents a possibly incomplete execution of transactions. As we mentioned in Chapter 1, we'll be interested in handling various types of failures, notably failures of the DBS. Such a failure may interrupt the execution of active transactions. Therefore our theory of executions must accommodate arbitrary histories, not merely complete ones. We draw histories as dags, employing the same conventions as for transactions.

To illustrate these definitions consider three transactions.

$$T_1 = r_1[x] \rightarrow w_1[x] \rightarrow c_1$$

$$T_3 = r_3[x] \rightarrow w_3[y] \rightarrow w_3[x] \rightarrow c_3$$

$$T_4 = r_4[y] \rightarrow w_4[x] \rightarrow w_4[y] \rightarrow w_4[z] \rightarrow c_4$$

A complete history over $\{T_1, T_3, T_4\}$ is

$$H_1 = \begin{tikzcd}
 & r_3[x] \arrow[r] & w_3[y] \arrow[r] & w_3[x] \arrow[r] & c_3 \\
r_4[y] \arrow[r] & w_4[x] \arrow[r] \arrow[u] & w_4[y] \arrow[r] \arrow[u] & w_4[z] \arrow[r] & c_4 \\
r_1[x] \arrow[r] & w_1[x] \arrow[r] \arrow[u] & c_1 & &
\end{tikzcd}$$

A history over the same three transactions (that also happens to be a prefix of H_1) is

$$H_1' = \begin{tikzcd}
 & r_3[x] \arrow[r] & w_3[y] \\
r_4[y] \arrow[r] & w_4[x] \arrow[r] \arrow[u] & w_4[y] \arrow[u] \\
r_1[x] \arrow[r] & w_1[x] \arrow[r] \arrow[u] & c_1
\end{tikzcd}$$

[4] See Section A.4 of the Appendix for the definition of prefix of a partial order.

As usual we do not draw all arrows implied by transitivity.

We will often deal with histories that are total orders of operations, such as:

$$r_1[x] \rightarrow r_3[x] \rightarrow w_1[x] \rightarrow c_1 \rightarrow w_3[y] \rightarrow w_3[x] \rightarrow c_3.$$

We will normally drop the arrows when writing such histories, as in:

$$r_1[x]\ r_3[x]\ w_1[x]\ c_1\ w_3[y]\ w_3[x]\ c_3.$$

A transaction T_i is *committed* (or *aborted*) in history H if $c_i \in H$ (or $a_i \in H$). T_i is *active* in H if it is neither committed nor aborted. Of course a complete history has no active transactions. Given a history H, the *committed projection of* H, denoted $C(H)$, is the history obtained from H by deleting all operations that do not belong to transactions committed in H.[5]

Note that $C(H)$ is a complete history over the set of committed transactions in H. If H represents an execution at some point in time, $C(H)$ is the only part of the execution we can count on, since active transactions can be aborted at any time, for instance, in the event of a system failure.

2.2 SERIALIZABLE HISTORIES

Histories represent concurrent executions of transactions. We are now ready to characterize serializable histories, that is, histories that represent serializable executions. Recall that an execution is serializable if it's equivalent to a serial execution of the same transactions. Our plan is to

□ define conditions under which two histories are equivalent;

□ define conditions under which a history is serial; and

□ define a history to be serializable if it is equivalent to a serial one.

Equivalence of Histories

We define two histories H and H' to be *equivalent* ($\equiv$) if

1. they are defined over the same set of transactions and have the same operations; and
2. they order conflicting operations of nonaborted transactions in the same way; that is, for any conflicting operations p_i and q_j belonging to transactions T_i and T_j (respectively) where $a_i, a_j \notin H$, if $p_i <_H q_j$ then $p_i <_{H'} q_j$.[6]

[5] More formally, $C(H)$ is the restriction of H on domain $\cup_{c_i \in H} T_i$ (cf. Section A.4 of the Appendix.)

[6] Note that this implies: $p_i <_H q_j$ iff $p_i <_{H'} q_j$ (why?).

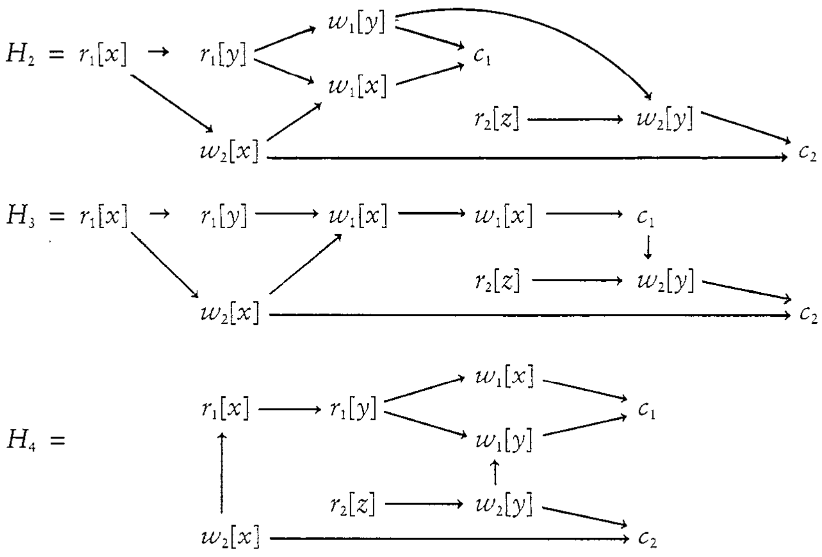


FIGURE 2–1
Example Histories
H2 and *H3* are equivalent, but *H4* is not equivalent to either.

The idea underlying this definition is that the outcome of a concurrent execution of transactions depends only on the relative ordering of conflicting operations. To see this observe that executing two nonconflicting operations in either order has the same computational effect. Conversely, the computational effect of executing two conflicting operations depends on their relative order.

For example, given the three histories shown in Fig. 2-1, $H_2 \equiv H_3$ but H_4 is not equivalent to either. (Keep in mind the orderings implied transitively by the arrows shown.)

Serializable Histories

A complete history H is *serial* if, for every two transactions T_i and T_j that appear in H, either all operations of T_i appear before all operations of T_j or vice versa. Thus, a serial history represents an execution in which there is no interleaving of the operations of different transactions. Each transaction executes from beginning to end before the next one can start.

We'll often denote a serial history over $\{T_1, T_2, .., T_n\}$ as $T_{i_1} T_{i_2} \ldots \mathrm{T}_{i_n}$ where $i_1, i_2, \ldots, i_n$ is a permutation of $1, 2, \ldots, n$. This means that T_{i_1} appears first in the serial history, T_{i_2} second, and so on.

At this point we would like to define a history H to be serializable if it is equivalent to some serial history H_s. This would be a perfectly reasonable definition if H were a *complete* history. Otherwise there are problems. First, there is an artificial problem in that a partial history can never be equivalent to a serial one. This is because serial histories must be complete by definition, and two histories can be equivalent only if they contain the same set of operations. But even assuming for the moment that we did not insist on serial histories being complete, we would still have a problem—a problem of substance and not mere definition. Namely, an incomplete execution of a transaction does not necessarily preserve the consistency of the database. Thus, serializability would be an inappropriate correctness condition if it merely stated that a history be computationally equivalent to a serial execution of some possibly partial transactions, since such a history does not necessarily represent a consistency preserving execution.

We are therefore led to a slightly more complex definition of serializability. Although more complex, the definition is still natural and, most importantly, sound. A history H is *serializable* (*SR*) if its committed projection, C(H), is equivalent to a serial history H_s.

This takes care of the problems, because C(H) is a *complete* history. Moreover, it is not an arbitrarily chosen complete history. If H represents the execution so far, it is really only the *committed* transactions whose execution the DBS has unconditionally guaranteed. All other transactions may be aborted.

2.3 THE SERIALIZABILITY THEOREM

We can determine whether a history is serializable by analyzing a graph derived from the history called a serialization graph. Let H be a history over $T = \{T_1, \ldots, T_n\}$. The *serialization graph* (*SG*) for H, denoted SG(H), is a directed graph whose nodes are the transactions in T that are committed in H and whose edges are all $T_i \rightarrow T_j$ $(i \neq j)$ such that one of T_i's operations precedes and conflicts with one of T_j's operations in H. For example:

$$H_5 = \begin{tikzcd} & & r_3[x] \arrow[r] & \mathrm{w}_3[x] \arrow[r] & c_3 \\ r_1[x] \arrow[r] & w_1[x] \arrow[r] \arrow[ur] & w_1[y] \arrow[r] & c_1 & \\ r_2[x] \arrow[r] \arrow[ur] & w_2[y] \arrow[r] \arrow[ur] & c_2 & & \end{tikzcd}$$

$$\mathrm{SG}(H_5) = \begin{tikzcd} T_2 \arrow[r] \arrow[rr, bend left] & T_1 \arrow[r] & T_3 \end{tikzcd}$$

The edge $T_1 \rightarrow T_3$ is in SG(H_5) because $w_1[x] < r_3[x]$, and the edge $T_2 \rightarrow T_3$ is in SG(H_5) because $r_2[x] < w_3[x]$. Notice that a single edge in SG(H_5) can be present because of more than one pair of conflicting operations. For instance, the edge $T_2 \rightarrow T_1$ is caused both by $r_2[x] < w_1[x]$ and $w_2[y] < w_1[y]$.

In general, the existence of edges $T_i \rightarrow T_j$ and $T_j \rightarrow T_k$ in an SG does not necessarily imply the existence of edge $T_i \rightarrow T_k$. For instance, with $w_3[z]$ replacing $w_3[x]$ in T_3, SG(H_5) becomes

$T_2 \rightarrow T_1 \rightarrow T_3$

since there is no conflict between T_2 and T_3.[7]

Each edge $T_i \rightarrow T_j$ in SG(H) means that at least one of T_i's operations precedes and conflicts with one of T_j's. This suggests that T_i should precede T_j in any serial history that is equivalent to H. If we can find a serial history, H_s, consistent with all edges in SG(H), then $H_s \equiv H$ and so H is SR. We can do this as long as SG(H) is acyclic.

In our previous example, SG(H_5) is acyclic. A serial history where transactions appear in an order consistent with the edges of SG(H_5) is T_2 T_1 T_3. Indeed this is the only such serial history. You can easily verify that H_5 is equivalent to T_2 T_1 T_3 and is therefore SR. We formalize this intuitive argument in the following theorem—the fundamental theorem of serializability theory.

Theorem 2.1: (The Serializability Theorem) A history H is serializable iff SG(H) is acyclic.

Proof: (if) Suppose H is a history over $T = \{T_1, T_2, ..., T_n\}$. Without loss of generality, assume $T_1, T_2, ..., T_m$ ($m \leq n$) are all of the transactions in T that are committed in H. Thus $T_1, T_2, ..., T_m$ are the nodes of SG(H). Since SG(H) is acyclic it may be topologically sorted.[8] Let $i_1, ..., i_m$ be a permutation of $1, 2, ..., m$ such that $T_{i_1}, T_{i_2}, ..., T_{i_m}$ is a topological sort of SG(H). Let H_s be the serial history T_{i_1} T_{i_2} ... T_{i_m}. We claim that C(H) $\equiv H_s$. To see this, let $p_i \in T_i$ and $q_j \in T_j$, where T_i, T_j are committed in H. Suppose p_i, q_j conflict and $p_i <_H q_j$. By definition of SG(H), $T_i \rightarrow T_j$ is an edge in SG(H). Therefore in any topological sort of SG(H), T_i must appear before T_j. Thus in H_s all operations of T_i appear before any operation of T_j, and in particular, $p_i <_{H_s} q_j$. We have proved that any two conflicting operations are ordered in C(H) in the same way as H_s. Thus C(H) $\equiv H_s$ and, because H_s is serial by construction, H is SR as was to be proved.

(only if) Suppose history H is SR. Let H_s be a serial history equivalent to C(H). Consider an edge $T_i \rightarrow T_j$ in SG(H). Thus there are two conflicting operations p_i, q_j of T_i, T_j (respectively), such that $p_i <_H q_j$. Because C(H) $\equiv H_s$, $p_i <_{H_s} q_j$. Because H_s is serial and p_i in T_i precedes q_j in T_j, it follows that T_i appears before T_j in H_s. Thus, we've shown that if $T_i \rightarrow T_j$ is in SG(H) then T_i appears before T_j in H_s. Now suppose there is a cycle in SG(H), and without loss of generality let that cycle be $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_k \rightarrow T_1$. These edges imply that in H_s, T_1 appears before T_2 which appears before T_3 which appears ... before T_k which appears before T_1. Thus, the existence of the cycle implies that each of $T_1, T_2, ..., T_k$ appears before

[7] We say that *two transactions conflict* if they contain conflicting operations.

[8] See Section A.3 of the Appendix for a definition of "topological sort of a directed acyclic graph."

itself in the serial history H_s, an absurdity. So no cycle can exist in SG(H). That is, SG(H) is an acyclic directed graph, as was to be proved. □

From the proof of the "if" part of this theorem we see that if a complete history H has an acyclic SG(H), then H is equivalent to *any* serial history that's a topological sort of SG(H). Since the latter can have more than one topological sort, H may be equivalent to more than one serial history. For instance,

$$H_6 = w_1[x]\ w_1[y]\ c_1\ r_2[x]\ r_3[y]\ w_2[x]\ c_2\ w_3[y]\ c_3$$

has the serialization graph

$$\text{SG}(H_6) = \begin{tikzcd} T_1 \arrow[r] \arrow[rr, bend left] & T_3 & T_2 \end{tikzcd}$$

which has two topological sorts, T_1, T_2, T_3 and T_1, T_3, T_2. Thus H_6 is equivalent to both $T_1\ T_2\ T_3$ and $T_1\ T_3\ T_2$.

2.4 RECOVERABLE HISTORIES

In Chapter 1, we saw that to ensure correctness in the presence of failures the scheduler must produce executions that are not only SR but also recoverable. We also discussed some additional requirements that may be desirable—preventing cascading aborts and the loss of before images—leading us to the idea of strict executions. Like serializability, these concepts can be conveniently formulated in the language of histories.

A transaction T_i reads data item x from T_j if T_j was the transaction that had last written into x but had not aborted at the time T_i read x. More precisely, we say that T_i *reads x from* T_j in history H if

1. $w_j[\mathrm{x}] < r_i[x\ ]$;
2. $a_j \nless r_i[x]$[9] and
3. if there is some $w_k[x]$ such that $w_j[x] < w_k[x] < r_i[x]$, then $a_k < r_i[x]$.

We say that T_i *reads from* T_j in H if T_i reads some data item from T_j in H. Notice that it is possible for a transaction to read a data item from itself (i.e., $w_i[x] < r_i[x]$).

A history H is called *recoverable* (RC) if, whenever T_i reads from T_j ($i \neq j$) in H and $c_i \in H$, $c_j < c_i$. Intuitively, a history is recoverable if each transaction commits after the commitment of all transactions (other than itself) from which it read.

A history H *avoids cascading aborts* (ACA) if, whenever T_i reads x from T_j ($i \neq j$), $c_j < r_i[x]$. That is, a transaction may read only those values that are written by committed transactions or by itself.

[9] $p \nless q$ denotes that operation p does not precede q in the partial order.

A history H is *strict* (ST) if whenever $w_j[x] < o_i[x]$ ($i \neq j$), either $a_j < o_i[x]$ or $c_j < o_i[x]$ where $o_i[x]$ is $r_i[x]$ or $w_i[x]$. That is, no data item may be read or overwritten until the transaction that previously wrote into it terminates, either by aborting or by committing.

We illustrate these definitions using the following four histories, over transactions:

$$T_1 = w_1[x]\ w_1[y]\ w_1[z]\ c_1 \qquad T_2 = r_2[u]\ w_2[x]\ r_2[y]\ w_2[y]\ c_2$$

$$H_7 = w_1[x]\ w_1[y]\ r_2[u]\ w_2[x]\ r_2[y]\ w_2[y]\ c_2\ w_1[z]\ c_1$$

$$H_8 = w_1[x]\ w_1[y]\ r_2[u]\ w_2[x]\ r_2[y]\ w_2[y]\ w_1[z]\ c_1\ c_2$$

$$H_9 = w_1[x]\ w_1[y]\ r_2[u]\ w_2[x]\ w_1[z]\ c_1\ r_2[y]\ w_2[y]\ c_2$$

$$H_{10} = w_1[x]\ w_1[y]\ r_2[u]\ w_1[z]\ c_1\ w_2[x]\ r_2[y]\ w_2[y]\ c_2$$

H_7 is not an RC (i.e., recoverable) history. To see this, note that T_2 reads y from T_1, but $c_2 < c_1$. H_8 *is* RC but not ACA (i.e., does not avoid cascading aborts), because T_2 reads y from T_1 before T_1 is committed. H_9 is ACA but not ST (i.e., strict), because T_2 overwrites the value written into x by T_1 before the latter terminates. H_{10} is ST.

In the remainder of this section, we will use RC, ACA, ST, and SR to denote the set of histories that are recoverable, avoid cascading aborts, are strict, and are serializable (respectively). Our next theorem says that recoverability, avoiding cascading aborts, and strictness are increasingly restrictive properties.

Theorem 2.2: ST $\subset$ ACA $\subset$ RC

Proof: Let $H \in$ ST. Suppose T_i reads x from T_j in H ($i \neq j$). Then we have $w_j[x] < r_i[x]$ and $a_j \not< r_i[x]$. Thus, by definition of ST, $c_j < r_i[x]$. Therefore $H \in$ ACA. This shows that ST $\subseteq$ ACA. History H_9 (above) avoids cascading aborts but isn't strict, implying ST $\neq$ ACA. Hence ST $\subset$ ACA.

Now let $H \in$ ACA, and suppose T_i reads x from T_j in H ($i \neq j$) and $c_i \in H$. Because H avoids cascading aborts, we must have $w_j[x] < c_j < r_i[x]$. Since $c_i \in H$, $r_i[x] < c_i$ and therefore $c_j < c_i$, proving $H \in$ RC. Thus ACA $\subseteq$ RC. History H_8 (above) is in RC but not in ACA, proving ACA $\neq$ RC. Hence ACA $\subset$ RC. □

SR intersects all of the sets RC, ACA, and ST, but is incomparable to each of them.[10] The relationships among the four sets are illustrated in Fig. 2–2 by a Venn diagram. The diagram shows where histories H_7 – H_{10} belong. All inclusions shown in Fig. 2–2 are proper (see Exercise 2.7).

[10] Two sets are *incomparable* if neither is contained in the other.

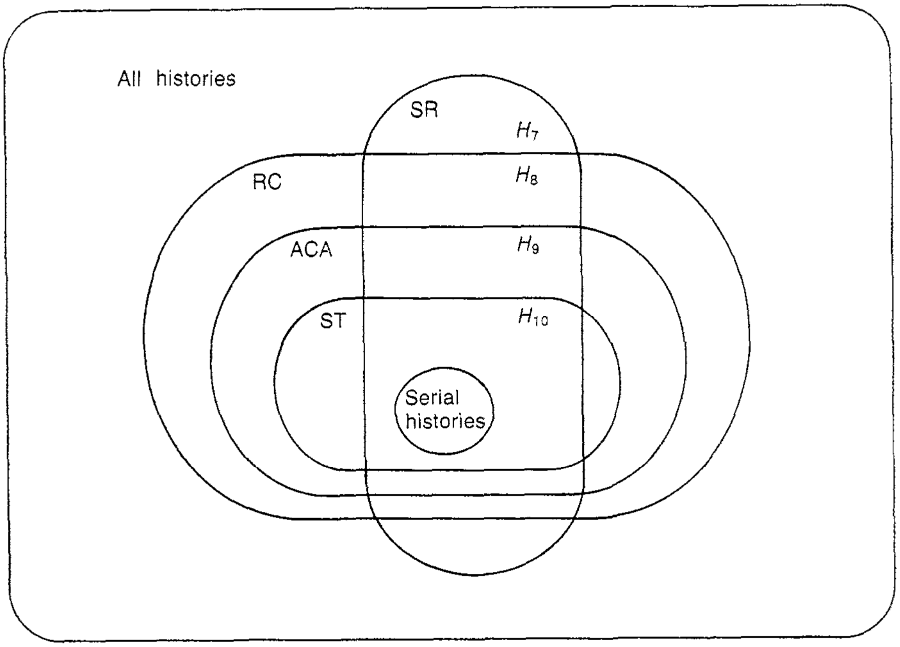


FIGURE 2–2
Relationships between Histories that are SR, RC, ACA, and ST

Figure 2–2 illustrates that there exist histories that are SR but not RC. In a DBS that must correctly handle transaction and system failures (as most must do), the scheduler must enforce recoverability (or the even stronger properties of cascadelessness or strictness) *in addition to* serializability.

We conclude this section with an important observation. A property of histories is called *prefix commit-closed* if, whenever the property is true of history H, it is also true of history C(H'), for any prefix H' of H. A correctness criterion for histories that accounts for transaction and system failures *must* be described by such a property. To see this, suppose that H is a "correct" history, that is, one that could be produced by a correct scheduler. Hence, any prefix H' of H could also be produced by the scheduler. But suppose at the time H' had been produced, the DBS failed. Then, at recovery time, the database should reflect the execution of exactly those transactions that were committed in H' that is, it should reflect the execution represented by C(H'). If the DBS is to handle transaction and system failures, C(H') had better be a "correct" history too.

It is easy to verify that RC, ACA, and ST are indeed prefix commit-closed properties (Exercise 2.9). As shown in the following theorem, serializability is too.

Theorem 2.3: Serializability is a prefix commit-closed property. That is, if H is an SR history, then for any prefix H' of H, $C(H')$ is also SR.

Proof: Since H is SR, SG(H) is acyclic (from the "only if" part of Theorem 2.1). Consider SG(C(H')) where H' is any prefix of H. If $T_i \rightarrow T_j$ is an edge of this graph, then we have two conflicting operations p_i, q_j belonging to T_i, T_j (respectively) with $p_i <_{C(H')} q_j$. But then clearly $p_i <_H q_j$ and thus the edge $T_i \rightarrow T_j$ exists in SG(H) as well. Therefore SG(C(H')) is a subgraph of SG(H). Since the latter is acyclic, the former must be too. By the "if" part of Theorem 2.1, it follows that C(H') is SR, as was to be proved. □

2.5 OPERATIONS BEYOND READS AND WRITES

In Chapter 1 we assumed that Read and Write are the only operations that transactions can perform on the database. However, neither the theoretical nor practical results presented in this book depend very much on this assumption. To help us understand how to treat a more general set of operations, let's reexamine serializability theory with new operations in mind.

Suppose we allow other database operations in addition to Read and Write. If transactions can interact through these operations, then these operations must appear in histories. Since every pair of conflicting operations in a history must be related by $<$, we must extend the definition of conflict to cover the new operations. The definition should be extended so that it retains the essence of conflict. Namely, two operations must be defined to conflict if, *in general*, the computational effect of their execution depends on the order in which they are processed.[11] The computational effect of the two operations consists of both the value returned by each operation (if any) and the final value of the data item(s) they access. If we extend the definition of conflict in this way, the definition of equivalent histories will remain valid in that only histories with the same computational effect will be defined to be equivalent.

The definition of SG remains unchanged. Moreover, since the proof of Theorem 2.1 only depends on the notion of conflict, not on the nature of the operations, it remains unchanged too. That is, a history is SR iff its SG is acyclic.

So, to add new operations in addition to Read and Write, the only work we need to do is to extend the definition of conflict.

For example, suppose we add Increment and Decrement to our repertoire of operations. Increment(x) adds 1 to data item x and Decrement(x) subtracts 1 from x. (This assumes, of course, that x's value is a number.) An Increment or Decrement does *not* return a value to the transaction that issued it. We abbreviate these operations by $inc_i[x]$ and $dec_i[\,x]$, where the subscript denotes the transaction that issued the operation. Since transactions can interact

[11] Note the qualification "in general" in this phrase. For example, one can think of *particular* Write operations that access the same data item and do not conflict, such as two Writes that write the same value. However, *in general*, two Writes on the same data item do conflict.

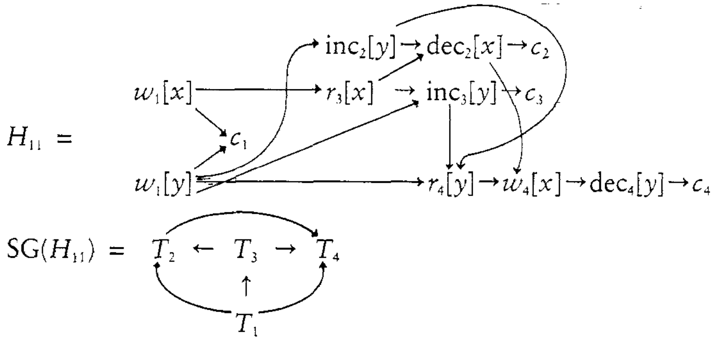
	Read	Write	Increment	Decrement
Read	y	n	n	n
Write	n	n	n	n
Increment	n	n	y	y
Decrement	n	n	y	y

FIGURE 2–3
A Compatibility Matrix

through Increments and Decrements (via Reads and Writes), Increments and Decrements must appear in histories.

We define two operations to *conflict* if they operate on the same data item and either at least one of them is a Write, or one is a Read and the other is an Increment or Decrement. We can conveniently express which combinations of operations conflict by a table called a *compatibility matrix*. The compatibility matrix for Read, Write, Increment, and Decrement is shown in Fig. 2–3. A "y" entry indicates that the operations in the corresponding row and column are compatible (i.e., do *not* conflict), while an "n" indicates that they are incompatible (i.e., conflict). Take a moment to convince yourself that the computational effect of executing two operations (as defined previously) depends on the order in which they were processed iff there is an "n" in the row and column combination corresponding to the operations.

A history that uses these operations is given below along with its SG.



Since SG(H_{11}) is acyclic, the generalized Serializability Theorem says that H_{11} is SR. It is equivalent to the serial history T_1 T_3 T_2 T_4, which can be obtained by topologically sorting SG(H_{11}).

2.6 VIEW EQUIVALENCE

This section explores another notion of history equivalence. To introduce it, let's rethink the concept from first principles.

We want to define equivalence so that two histories are equivalent if they have the same effects. The effects of a history are the values produced by the Write operations of unaborted transactions.

Since we don't know anything about the computation that each transaction performs, we don't know much about the value written by each Write. All we *do* know is that it is *some* function of the values read by each of the transaction's Reads that preceded the Write. Thus, if each transaction's Reads read the same value in two histories, then its Writes will produce the same values in both histories. From this observation and a little careful thought, we can see that (1) if each transaction reads each of its data items from the same Writes in both histories, then all Writes write the same values in both histories, and (2) if for each data item x, the final Write on x is the same in both histories, then the final value of all data items will be the same in both histories. And if all Writes write the same values in both histories and leave the database in the same final state, then the histories must be equivalent.

This leads us to the following definition of history equivalence. The *final write* of x in a history H is the operation $w_i[x] \in H$, such that $a_i \notin H$ and for any $w_j[x] \in H$ $(j \neq i)$ either $w_j[x] < w_i[x]$ or $a_j \in H$. Two histories H, H' are *equivalent* if

1. they are over the same set of transactions and have the same operations;
2. for any T_i, T_j such that $a_i, a_j \in H$ (hence $a_i, a_j \notin H'$) and for any x, if T_i reads x from T_j in H then T_i reads x from T_j in H' and
3. for each x, if $w_i[x]$ is the final write of x in H then it is also the final write of x in H'.

As we'll see later, this definition of equivalence is somewhat different from the one we've used so far, so to distinguish it we'll give it a different name, *view equivalence* (since, by (2), each Read has the same view in both histories). For clarity, we'll call the old definition *conflict equivalence* (since it says that two histories are equivalent if conflicting operations of unaborted transactions appear in the same order in both histories).

View equivalence will be very useful in Chapters 5 and 8 for our formal treatment of concurrency control algorithms for multicopy data.

*View Serializability[12]

In Section 2.2 we defined a history H to be serializable if its committed projection, C(H), is (conflict) equivalent to some serial history. We can use the definition of view equivalence in a similar manner to arrive at a new concept of

[12] The asterisk before this section title means that the rest of this section goes deeper into serializability theory than is needed to understand succeeding nontheoretical sections (cf. the Preface).

serializability. Specifically, we define a history H to be *view serializable* (*VSR*) if for any prefix H' of H, $C(H')$ is view equivalent to some serial history. For emphasis we'll use *conflict serializable* (*CSR*) for what we have thus far simply called serializable.

The reason for insisting that the committed projection of *every prefix* of H (instead of just the committed projection of H itself) be view equivalent to a serial history is to ensure that view serializability is a prefix commit-closed property. Consider, for instance,

$$H_{12} = w_1[x]\ w_2[x]\ w_2[y]\ c_2\ w_1[y]\ c_1\ w_3[x]\ w_3[y]\ c_3.$$

$C(H_{12}) = H_{12}$ and is view equivalent to $T_1\ T_2\ T_3$. However, if we take H'_{12} to be the prefix of H_{12} up to and including c_1, we have that $C(H'_{12})$ is not view equivalent to either $T_1\ T_2$ or $T_2\ T_1$. Thus we wouldn't get a prefix commit-closed property if we had defined view serializability by requiring only that the committed projection of that history itself be equivalent to a serial history.[13] As discussed at the end of Section 2.2, this would make view serializability an inappropriate correctness criterion in an environment where transactions or the system are subject to failures.

The two "versions" of serializability are not the same. In fact, as the following theorem shows, view serializability is a (strictly) more inclusive concept.

Theorem 2.4: If H is conflict serializable then it is view serializable. The converse is not, generally, true.

Proof: Suppose H is conflict serializable. Consider an arbitrary prefix H' of H. By assumption, $C(H')$ is conflict equivalent to some serial history, say H_s. We claim that $C(H')$ is view equivalent to H_s. Clearly, $C(H')$ and H_s are over the same set of transactions and have the same operations (since they are conflict equivalent). It remains to show that $C(H')$ and H_s have the same reads-from relationships and final writes for all data items. Suppose T_i reads x from T_j in $C(H')$. Then, $w_j[x] <_{C(H')} r_i[x]$ and there is no $w_k[x]$ such that $w_j[x] <_{C(H')} w_k[x] <_{C(H')} r_i[x]$. Because $w_j[x]$, $r_i[x]$ conflict with each other and $w_k[x]$ conflicts with both, and because $C(H')$ and H_s order conflicting operations in the same way, it follows that $w_j[x] <_{H_s} r_i[x]$ and there is no $w_k[x]$ such that $w_j[x] <_{H_s} w_k[x] <_{H_s} r_i[x]$. Hence T_i reads x from T_j in H_s. If T_i reads x from T_j in H_s, then the same argument implies T_i reads x from T_j in $C(H')$. Thus $C(H')$ and H_s have the same reads-from relationships. Because Writes on the same data item conflict and $C(H')$ and H_s order conflicting operations in the same way, the

[13] In case of *conflict* serializability, however, requiring that the committed projection of the history be conflict equivalent to a serial history was sufficient to ensure prefix commit closure (see Theorem 2.3).

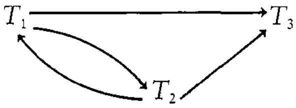
two histories must also have the same final write for each data item. C(H') and H_s are therefore view equivalent. Since H' is an arbitrary prefix of H, it follows that H' is view serializable.

To show that the converse is not, generally, true, consider

$$H_{13} = w_1[x]\ w_2[x]\ w_2[y]\ c_2\ w_1[y]\ w_3[x]\ w_3[y]\ c_3\ \mathrm{w}_1[z]\ c_1.$$

H_{13} is view serializable. To see this, consider any prefix H'_{13} of H_{13}. If H'_{13} includes c_1 (i.e. $H'_{13} = H_{13}$, it is view equivalent to $T_1\ T_2\ T_3$; if H'_{13} includes c_3 but not c_1, it is view equivalent to $T_2\ T_3$; if H'_{13} includes c_2 but not c_3, it is view equivalent to T_2; finally, if H'_{13} does not include c_2 it is view equivalent to the empty serial history. However, H_{13} is not conflict serializable because its SG, shown below, has a cycle.

SG(H_{13}) =



□

Even though view serializability is more inclusive than conflict serializability, there are reasons for keeping the latter as our concurrency control correctness criterion. From a practical point of view, all known concurrency control algorithms are *conflict based*. That is, their goal is to order *conflicting* operations in a consistent way, and as a result produce only conflict serializable histories. From a theoretical standpoint, it is hopeless to expect efficient schedulers to be based on view serializability. Technically, it can be shown that an efficient scheduler that produces exactly the set of all view serializable histories can only exist if the famous $P = NP$? problem has an affirmative answer. This is considered very unlikely, as it would imply that a wide variety of notoriously difficult combinatorial problems would be solvable by efficient algorithms.

Thus, in the rest of the book we continue to use the terms equivalent and serializable (histories) to mean *conflict* equivalent and *conflict* serializable (histories), unless otherwise qualified.

BIBLIOGRAPHIC NOTES

Virtually all rigorous treatments of concurrency control use some form of the history model of executions. See [Bernstein, Shipman, Wong 79], [Gray et al. 75], [Papadimitriou 79], and [Stearns, Lewis, Rosenkrantz 76]. An extensive treatment of serializability theory appears in [Papadimitriou 86].

The definition of equivalence and serializability used here and the Serializability Theorem are from [Gray et al. 75]. View equivalence and view serializability are defined in [Yannakakis 84]. Recoverability, avoidance of cascading aborts, and strictness are defined in [Hadzilacos 83] and [Hadzilacos 86]. See also [Papadimitriou, Yannakakis 85].

[Papadimitriou 79] proves that no efficient scheduler can output all view serializable histories, unless $P = NP$. (For the $P = NP$? question see [Garey, Johnson 79].)

EXERCISES

2.1 Let H_1 and H_2 be *totally ordered* histories. Define a (symmetric) relation $\sim$ between such histories as follows:

$$H_1 \sim H_2 \text{ iff } \begin{aligned} H_1 &= p_1 p_2 \ldots p_{i-1}\, p_i\, p_{i+1} \ldots p_n \text{ and} \\ H_2 &= p_1 p_2 \ldots p_{i-1}\, p_{i+1}\, p_i\, p_{i+2} \ldots p_n \end{aligned}$$

where p_i, p_{i+1} are operations such that either the operations do not conflict or (at least) one of the two transactions issuing the operations is aborted in H_1 (and hence in H_2). Extend $\sim$ to arbitrary (not necessarily totally ordered) histories as follows:

$H_1 \sim H_2$ iff there exist totally ordered histories
H'_1, H'_2 compatible with H_1, H_2 (respectively), such that $H'_1 \sim H'_2$.

(H' is compatible with H if they have the same operations and $p <_H q$ implies $p <_{H'} q$). Finally, define $\approx$ to be the transitive closure of $\sim$. That is,

$H \approx H'$ iff there exist histories $H_1, H_2, \ldots, H_k$ ($k \geq 1$) such that
$H = H_1 \sim H_2 \sim \ldots \sim H_k = H'$.

Prove that $H' \equiv H$ iff $H \approx H'$.

2.2 Prove that if two histories are equivalent then their serialization graphs are identical.

2.3 The converse of Exercise 2.2 is obviously not true. For example, the histories $w_1[x]\, w_2[x]$ and $w_1[y]\, w_2[y]$ have the same serialization graph but are clearly not equivalent. Is the following, stronger version of the converse in Exercise 2.2 true: If histories H and H' are over the same set of transactions, have the same operations, and have the same serialization graphs, then $H \equiv H'$.

2.4*[14] A *blind write* is a Write on some data item x by a transaction that did not previously read x. Suppose we insist that transactions have no blind writes. Formally, this means replacing condition (4) in the definition of transaction (cf. Section 2.1) by the stronger

(4′) if $w_i[x] \in T_i$ then $r_i[x] \in T_i$ and $r_i[x] <_i w_i[x]$.

Prove that under this assumption a history is view serializable iff it is conflict serializable.

[14] Starred exercises are not necessarily difficult, but require knowledge of starred sections in the text.

2.5* Prove that if H and H' are complete, recoverable, and view equivalent histories then they are conflict equivalent. Also prove that the requirements "complete and recoverable" are necessary for the truth of this statement.

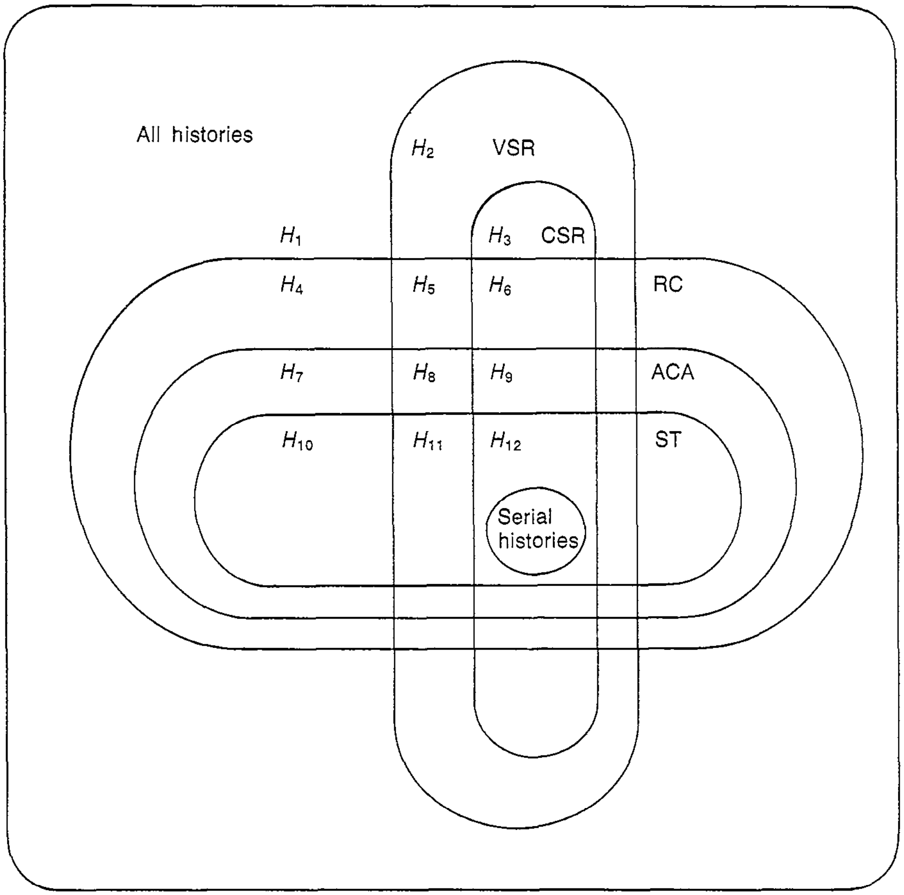
2.6* A *view graph* of history H is a directed graph with nodes corresponding to the committed transactions in H and edges defined as follows:

a. if T_i, T_j are committed transactions such that T_j reads from T_i then $T_i \rightarrow T_j$ is an edge in the view graph, and

b. if T_i, T_j, T_k are committed transactions, T_j reads x from T_i and T_k writes x then either $T_k \rightarrow T_i$ or $T_j \rightarrow T_k$ is an edge in the view graph.

As can be seen from (b), there may be several view graphs corresponding to the history H. Prove that H is view serializable iff for every prefix H' of H, C(H') has some acyclic view graph.

2.7* The following Venn diagram summarizes the relationships among the five sets of histories defined in Chapter 2.



Prove that all regions in the diagram represent non-empty sets by providing example histories $H_1 - H_{12}$.

2.8 In a draft of a proposed standard on commitment, concurrency and recovery (ISO, "Information Processing—Open Systems Interconnection—Definition of Common Application Service Elements—Part 3: Commitment, Concurrency and Recovery," Draft International Standard ISO/DIS 8649/3) the following definition of concurrency control correctness is given (p. 6):

> Concurrency control [ensures] that an atomic action is not committed unless
>
> 1. all atomic actions which have changed the value of a datum prior to its period of use by this atomic action have committed; and
> 2. no change has been made to the value of a datum during its period of use, except by [...] this atomic action.

The *period of use* of a datum by an atomic action is defined as "the time from first use of the datum by [...] the atomic action to the last use of that datum by the atomic action."

In our terminology an atomic action is a transaction and a datum is a data item.

a. What did we call condition (1) in this chapter?
b. Express this definition of concurrency control as conditions on histories.
c. How does the above, definition of concurrency control relate to serializability and recoverability? (Is it equivalent, stronger, weaker or unrelated to them?)
d. Suppose we define the period of use of a datum by an atomic action as the time from the first use of the datum by the atomic action to the time of the action's commitment or abortion. How does this change affect your answer to (c)?

2.9 Prove that RC, ACA, and ST are prefix commit-closed properties.

2.10 Redefine the concept of "transaction" in serializability theoretic terms so that each transaction T_i can read or write a data item more than once. To do this, you need to distinguish two Reads or Writes by T_i, say by another subscript, such as $r_{i1}[x]$ and $r_{i2}[x]$. Using this modified definition of "transaction," redefine each of the following terms if necessary: complete history, history, equivalence of histories, and serialization graph. Prove Theorem 2.1, the Serializability Theorem, for this new model.

2.11 Using the modified definition of "transaction" in the previous problem, redefine the concept of view equivalence and prove Theorem 2.4, that conflict serializable histories are view serializable.

2.12 Two transactions are *not interleaved* in a history if every operation of one transaction precedes every operation of the other. Give an example of a serializable history H that has all of the following properties:

a. transactions T_1 and T_2 are not interleaved in H;
b. T_1 precedes T_2 in H; and
c. in any serial history equivalent to H, T_2 precedes T_1.

The history may include more than two transactions.

2.13 Prove or disprove that every history H having the following property is serializable: If p_i, $q_j \in H$ and p_i, q_j conflict, then T_i and T_j are not interleaved in H. (See Exercise 2.12 for the definition of interleaved.)