19.2 View Serializability

Recall our discussion in Section ?? of how our true goal in the design of a scheduler is to allow only schedules that are serializable. We also saw how differences in what operations transactions apply to the data can affect whether or not a given schedule is serializable. We also learned in Section ?? that schedulers normally enforce "conflict serializability," which guarantees serializability regardless of what the transactions do with their data.

However, there are weaker conditions than conflict-serializability that also guarantee serializability. In this section we shall consider one such condition, called "view-serializability." Intuitively, view-serializability considers all the connections between transactions T and U such that T writes a database element whose value U reads. The key difference between view- and conflict-serializability appears when a transaction T writes a value A that no other transaction reads (because some other transaction later writes its own value for A). In that case, the $w_T(A)$ action can be placed in certain other positions of the schedule (where A is likewise never read) that would not be permitted under the definition of conflict-serializability. In this section, we shall define view-serializability precisely and give a test for it.

19.2.1 View Equivalence

Suppose we have two schedules S_1 and S_2 of the same set of transactions. Imagine that there is a hypothetical transaction T_0 that wrote initial values for each database element read by any transaction in the schedules, and another hypothetical transaction T_f that reads every element written by one or more transactions after each schedule ends. Then for every read action $r_i(A)$ in one of the schedules, we can find the write action $w_j(A)$ that most closely preceded the read in question.[1] We say T_j is the *source* of the read action $r_i(A)$. Note that transaction T_j could be the hypothetical initial transaction T_0, and T_i could be T_f.

If for every read action in one of the schedules, its source is the same in the other schedule, we say that S_1 and S_2 are *view-equivalent.* Surely, view-equivalent schedules are truly equivalent; they each do the same when executed on any one database state. If a schedule S is view-equivalent to a serial schedule, we say S is *view-serializable.*

Example 19.1: Consider the schedule S defined by:

T_1:			$r_1(A)$		$w_1(B)$		
T_2:	$r_2(B)$	$w_2(A)$				$w_2(B)$	
T_3:				$r_3(A)$			$w_3(B)$

[1] While we have not previously prevented a transaction from writing an element twice, there is generally no need for it to do so, and in this study it is useful to assume that a transaction only writes a given element once.

Notice that we have separated the actions of each transaction vertically, to indicate better which transaction does what; you should read the schedule from left-to-right, as usual.

In S, both T_1 and T_2 write values of B that are lost; only the value of B written by T_3 survives to the end of the schedule and is "read" by the hypothetical transaction T_f. S is not conflict-serializable. To see why, first note that T_2 writes A before T_1 reads A, so T_2 must precede T_1 in a hypothetical conflict-equivalent serial schedule. The fact that the action $w_1(B)$ precedes $w_2(B)$ also forces T_1 to precede T_2 in any conflict-equivalent serial schedule. Yet neither $w_1(B)$ nor $w_2(B)$ has any long-term affect on the database. It is these sorts of irrelevant writes that view-serializability is able to ignore, when determining the true constraints on an equivalent serial schedule.

More precisely, let us consider the sources of all the reads in S:

1. The source of $r_2(B)$ is T_0, since there is no prior write of B in S.
2. The source of $r_1(A)$ is T_2, since T_2 most recently wrote A before the read.
3. Likewise, the source of $r_3(A)$ is T_2.
4. The source of the hypothetical read of A by T_f is T_2.
5. The source of the hypothetical read of B by T_f is T_3, the last writer of B.

Of course, T_0 appears before all real transactions in any schedule, and T_f appears after all transactions. If we order the real transactions (T_2, T_1, T_3), then the sources of all reads are the same as in schedule S. That is, T_2 reads B, and surely T_0 is the previous "writer." T_1 reads A, but T_2 already wrote A, so the source of $r_1(A)$ is T_2, as in S. T_3 also reads A, but since the prior T_2 wrote A, that is the source of $r_3(A)$, as in S. Finally, the hypothetical T_f reads A and B, but the last writers of A and B in the schedule (T_2, T_1, T_3) are T_2 and T_3 respectively, also as in S. We conclude that S is a view-serializable schedule, and the schedule represented by the order (T_2, T_1, T_3) is a view-equivalent schedule. □

19.2.2 Polygraphs and the Test for View-Serializability

There is a generalization of the precedence graph, which we used to test conflict serializability in Section ??, that reflects all the precedence constraints required by the definition of view serializability. We define the *polygraph* for a schedule to consist of the following:

1. A node for each transaction and additional nodes for the hypothetical transactions T_0 and T_f.
2. For each action $r_i(X)$ with source T_j, place an arc from T_j to T_i.

3. Suppose T_j is the source of a read $r_i(X)$, and T_k is another writer of X. It is not allowed for T_k to intervene between T_j and T_i, so it must appear either before T_j or after T_i. We represent this condition by an *arc pair* (shown dashed) from T_k to T_j and from T_i to T_k. Intuitively, one or the other of an arc pair is "real," but we don't care which, and when we try to make the polygraph acyclic, we can pick whichever of the pair helps to make it acyclic. However, there are important special cases where the arc pair becomes a single arc:

   (a) If T_j is T_0, then it is not possible for T_k to appear before T_j, so we use an arc $T_i \rightarrow T_k$ in place of the arc pair.

   (b) If T_i is T_f, then T_k cannot follow T_i, so we use an arc $T_k \rightarrow T_j$ in place of the arc pair.

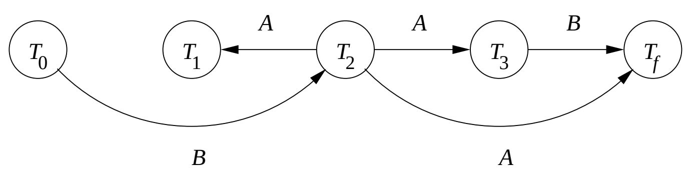

Figure 19.1: Beginning of polygraph for Example 19.2

Example 19.2: Consider the schedule S from Example 19.1. We show in Fig. 19.1 the beginning of the polygraph for S, where only the nodes and the arcs from rule (2) have been placed. We have also indicated the database element causing each arc. That is, A is passed from T_2 to T_1, T_3, and T_f, while B is passed from T_0 to T_2 and from T_3 to T_f.

Now, we must consider what transactions might interfere with each of these five connections by writing the same element between them. These potential interferences are ruled out by the arc pairs from rule (3), although as we shall see, in this example each of the arc pairs involves a special case and becomes a single arc.

Consider the arc $T_2 \rightarrow T_1$ based on element A. The only writers of A are T_0 and T_2, and neither of them can get in the middle of this arc, since T_0 cannot move its position, and T_2 is already an end of the arc. Thus, no additional arcs are needed. A similar argument tells us no additional arcs are needed to keep writers of A outside the arcs $T_2 \rightarrow T_3$ and $T_2 \rightarrow T_f$.

Now consider the arcs based on B. Note that T_0, T_1, T_2, and T_3 all write B. Consider the arc $T_0 \rightarrow T_2$ first. T_1 and T_3 are other writers of B; T_0 and T_2 also write B, but as we saw, the arc ends cannot cause interference, so we need not consider them. As we cannot place T_1 between T_0 and T_2, in principle we need the arc pair ($T_1 \rightarrow T_0$, $T_2 \rightarrow T_1$). However, nothing can precede T_0, so the option $T_1 \rightarrow T_0$ is not possible. We may in this special case just add the arc $T_2 \rightarrow T_1$ to the polygraph. But this arc is already there because of A, so in effect, we make no change to the polygraph to keep T_1 outside the arc $T_0 \rightarrow T_2$.

We also cannot place T_3 between T_0 and T_2. Similar reasoning tells us to add the arc $T_2 \rightarrow T_3$, rather than an arc pair. However, this arc too is already in the polygraph because of A, so we make no change.

Next, consider the arc $T_3 \rightarrow T_f$. Since T_0, T_1, and T_2 are other writers of B, we must keep them each outside the arc. T_0 cannot be moved between T_3 and T_f, but T_1 or T_2 could. Since neither could be moved after T_f, we must constrain T_1 and T_2 to appear before T_3. There is already an arc $T_2 \rightarrow T_3$, but we must add to the polygraph the arc $T_1 \rightarrow T_3$. This change is the only arc we must add to the polygraph, whose final set of arcs is shown in Fig. 19.2. □

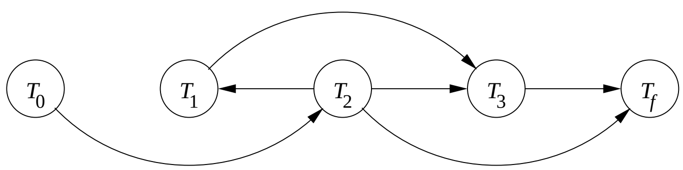

Figure 19.2: Complete polygraph for Example 19.2

Example 19.3: In Example 19.2, all the arc pairs turned out to be single arcs as a special case. Figure 19.3 is an example of a schedule of four transactions where there is a true arc pair in the polygraph.

T_1	T_2	T_3	T_4
	$r_2(A)$;		
$r_1(A)$; $w_1(C)$;			
		$r_3(C)$;	
$w_1(B)$;			
			$r_4(B)$;
		$w_3(A)$;	
			$r_4(C)$;
	$w_2(D)$; $r_2(B)$;		
			$w_4(A)$; $w_4(B)$;

Figure 19.3: Example of transactions whose polygraph requires an arc pair

Figure 19.4 shows the polygraph, with only the arcs that come from the source-to-reader connections. As in Fig. 19.1 we label each arc by the element(s) that require it. We must then consider the possible ways that arc pairs could be added. As we saw in Example 19.2, there are several simplifications we can make. When avoiding interference with the arc $T_j \rightarrow T_i$, the only transactions that need be considered as T_k (the transaction that cannot be in the middle) are:

- Writers of an element that caused this arc $T_j \rightarrow T_i$,
- But not T_0 or T_f, which can never be T_k, and
- Not T_i or T_j, the ends of the arc itself.

With these rules in mind, let us consider the arcs due to database element A, which is written by T_0, T_3, and T_4. We need not consider T_0 at all. T_3 must not get between $T_4 \rightarrow T_f$, so we add arc $T_3 \rightarrow T_4$; remember that the other arc in the pair, $T_f \rightarrow T_3$ is not an option. Likewise, T_3 must not get between $T_0 \rightarrow T_1$ or $T_0 \rightarrow T_2$, which results in the arcs $T_1 \rightarrow T_3$ and $T_2 \rightarrow T_3$.

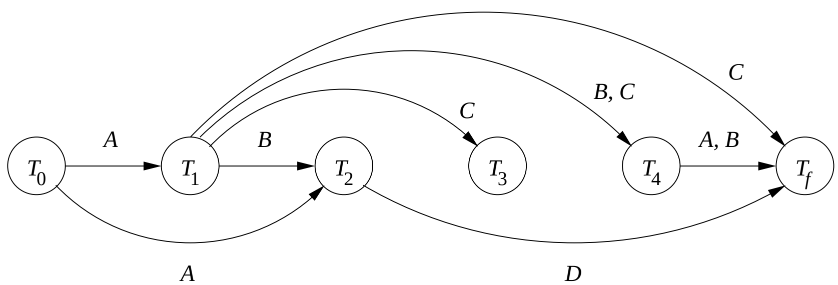

Figure 19.4: Beginning of polygraph for Example 19.3

Now, consider the fact that T_4 also must not get in the middle of an arc due to A. It is an end of $T_4 \rightarrow T_f$, so that arc is irrelevant. T_4 must not get between $T_0 \rightarrow T_1$ or $T_0 \rightarrow T_2$, which results in the arcs $T_1 \rightarrow T_4$ and $T_2 \rightarrow T_4$.

Next, let us consider the arcs due to B, which is written by T_0, T_1, and T_4. Again we need not consider T_0. The only arcs due to B are $T_1 \rightarrow T_2$, $T_1 \rightarrow T_4$, and $T_4 \rightarrow T_f$. T_1 cannot get in the middle of the first two, but the third requires arc $T_1 \rightarrow T_4$.

T_4 can get in the middle of $T_1 \rightarrow T_2$ only. This arc has neither end at T_0 or T_f, so it really requires an arc pair: $(T_4 \rightarrow T_1,\ T_2 \rightarrow T_4)$. We show this arc pair, as well as all the other arcs added, in Fig. 19.5.

Next, consider the writers of C: T_0 and T_1. As before, T_0 cannot present a problem. Also, T_1 is part of every arc due to C, so it cannot get in the middle. Similarly, D is written only by T_0 and T_2, so we can determine that no more arcs are necessary. The final polygraph is thus the one in Fig. 19.5. □

19.2.3 Testing for View-Serializability

Since we must choose only one of each arc pair, we can find an equivalent serial order for schedule S if and only if there is some selection from each arc pair that turns S's polygraph into an acyclic graph. To see why, notice that if there

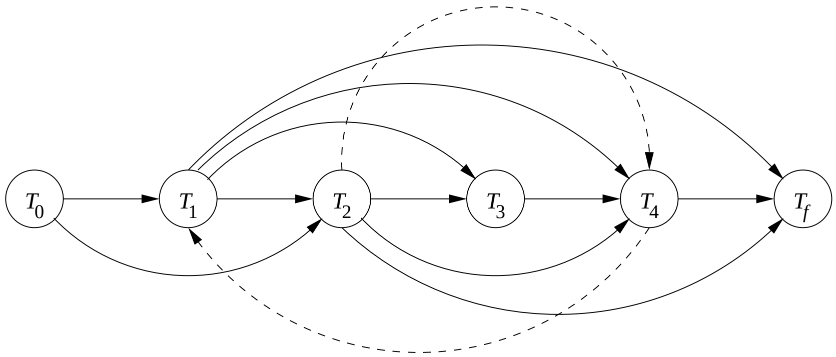

Figure 19.5: Complete polygraph for Example 19.3

is such an acyclic graph, then any topological sort of the graph gives an order in which no writer may appear between a reader and its source, and every writer appears before its readers. Thus, the reader-source connections in the serial order are exactly the same as in S; the two schedules are view-equivalent, and therefore S is view-serializable.

Conversely, if S is view-serializable, then there is a view-equivalent serial order S'. Every arc pair ($T_k \rightarrow T_j$, $T_i \rightarrow T_k$) in S's polygraph must have T_k either before T_j or after T_i in S'; otherwise the writing by T_k breaks the connection from T_j to T_i, which means that S and S' are not view-equivalent. Likewise, every arc in the polygraph must be respected by the transaction order of S'. We conclude that there is a choice of arcs from each arc pair that makes the polygraph into a graph for which the serial order S' is consistent with each arc of the graph. Thus, this graph is acyclic.

Example 19.4: Consider the polygraph of Fig. 19.2. It is already a graph, and it is acyclic. The only topological order is (T_2, T_1, T_3), which is therefore a view-equivalent serial order for the schedule of Example 19.2.

Now consider the polygraph of Fig. 19.5. We must consider each choice from the one arc pair. If we choose $T_4 \rightarrow T_1$, then there is a cycle. However, if we choose $T_2 \rightarrow T_4$, the result is an acyclic graph. The sole topological order for this graph is (T_1, T_2, T_3, T_4). This order yields a view-equivalent serial order and shows that the original schedule is view-serializable. □

19.2.4 Exercises for Section 19.2

Exercise 19.2.1: Draw the polygraph and find all view-equivalent serial orders for the following schedules:

* a) $r_1(A)$; $r_2(A)$; $r_3(A)$; $w_1(B)$; $w_2(B)$; $w_3(B)$;

b) $r_1(A)$; $r_2(A)$; $r_3(A)$; $r_4(A)$; $w_1(B)$; $w_2(B)$; $w_3(B)$; $w_4(B)$;

c) $r_1(A)$; $r_3(D)$; $w_1(B)$; $r_2(B)$; $w_3(B)$; $r_4(B)$; $w_2(C)$; $r_5(C)$; $w_4(E)$; $r_5(E)$; $w_5(B)$;

d) $w_1(A)$; $r_2(A)$; $w_3(A)$; $r_4(A)$; $w_5(A)$; $r_6(A)$;

! **Exercise 19.2.2:** Below are some serial schedules. Tell how many schedules are (*i*) conflict-equivalent and (*ii*) view-equivalent to these serial schedules.

* a) $r_1(A)$; $w_1(B)$; $r_2(A)$; $w_2(B)$; $r_3(A)$; $w_3(B)$; that is, three transactions each read A and then write B.

b) $r_1(A)$; $w_1(B)$; $w_1(C)$; $r_2(A)$; $w_2(B)$; $w_2(C)$; that is, two transactions each read A and then write B and C.