Conflict-Serializable Transactions:

1. Define transaction:

   This transaction updates the database when a delivery person completes a delivery request.

   Insert(Completed_Delivery)
   Insert(Past_Order)
   Write(DeliveryPerson.Status = 0)
   (*This indicates that the delivery person is now available.)*
   Delete(ActiveDeliveryRequest)
   Commit

   Consider two transactions T1 and T2 both of the above type representing two different delivery people completing deliveries concurrently.

T1	T2
Insert(Completed_Delivery) Insert(Past_Order)	
	Insert(Completed_Delivery) Insert(Past_Order)
Write(DeliveryPerson.Status=0)	
	Write(DeliveryPerson.Status=0)
Delete(ActiveDeliveryRequest) Commit	
	Delete(ActiveDeliveryRequest) Commit

   This schedule is Conflict-serializable since a series of swaps starting from the later steps in the transaction can produce a serial schedule of the form:

T1	T2
Insert(Completed_Delivery) Insert(Past_Order) Write(DeliveryPerson.Status=0) Delete(ActiveDeliveryRequest) Commit	Insert(Completed_Delivery) Insert(Past_Order) Write(DeliveryPerson.Status=0) Delete(ActiveDeliveryRequest) Commit

2. Define transaction:

   This transaction updates the database when a delivery person accepts a new delivery request.

   Write(DeliveryPerson.ActiveRequest = NewOrderID)
   (This indicates the delivery person is unavailable)
   Insert(ActiveDeliveryRequest)
   Write(DeliveryRequest.Status=1)
   Delete(PendingDeliveryRequest)
   Commit

   Consider two transactions T1 and T2 both of the above type representing two different delivery people accepting delivery requests concurrently.

T1	T2
Write(DeliveryPerson.ActiveRequest = NewOrderID) Insert(ActiveDeliveryRequest) Write(DeliveryRequestStatus=1)	Write(DeliveryPerson.ActiveRequest = NewOrderID) Insert(ActiveDeliveryRequest)

Delete(PendingDeliveryRequest) Commit	Write(DeliveryRequestStatus=1) Delete(PendingDeliveryRequest) Commit

This schedule is Conflict-serializable since a series of swaps starting from the later steps in the transaction can produce a serial schedule of the form:

T1	T2
Write(DeliveryPerson.ActiveRequest = NewOrderID) Insert(ActiveDeliveryRequest) Write(DeliveryRequestStatus=1) Delete(PendingDeliveryRequest) Commit	 Write(DeliveryPerson.ActiveRequest = NewOrderID) Insert(ActiveDeliveryRequest) Write(DeliveryRequestStatus=1) Delete(PendingDeliveryRequest) Commit

Non-Conflict Serializable:

Consider a schedule which tracks updates on the database upon completion of a delivery by a delivery person and the subsequent allocation of a new delivery request to them. This can be illustrated by rewriting the two transactions defined above. To do so, relabel the data objects as follows:

- Completed_Delivery: CD
- Past_Order: PO
- DeliveryRequest.Status: DRS
- ActiveDeliveryRequest: ADR
- DeliveryPerson.ActiveRequest: DPAR
- PendingDeliveryRequest: PDR

The database objects common to the two transactions are DeliveryRequest.Status and ActiveDeliveryRequest. Updated form of the two transactions in terms of read, write operations:

T1:

```
Write(CD)
Write(PO)
Read(DRS)
Write(DRS = 2)      //to indicate the request was completed
Write(ADR)
```

T2:

```
Write(DPAR = NewOrderID)
Read(DRS)
Write(DRS = 1)
Write(ADR)
Write(PDR)
```

Both T1->T2 and T2->T1 are possible in a real life context and are both serial schedules.

Consider the sequence of operations,

W1(CD), W1(PO), W2(DPAR = NewOrderID), R2(DRS), W2(DRS = 1), R1(DRS), W1(DRS = 2), W1(ADR), W2(ADR), W2(PDR).

It can be confirmed that there is no way of swapping the non-conflicting operations that turn this sequence into one of the serial schedules. Thus, this is a non-conflict serializable schedule.

T1	T2
Write(CD) Write(PO)	Write(DPAR = NewOrderID) Read(DRS) Write(DRS = 1)
Read(DRS) Write(DRS = 2) Write(ADR)	Write(ADR) Write(PDR)