
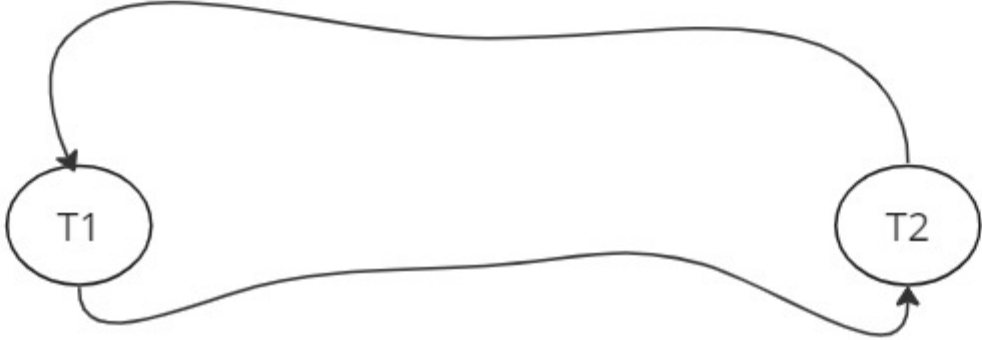


CONFLICTING TRANSACTIONAL QUERIES

	Transactional Query T1.	Transactional Query T2.	Remarks
Query:	start transaction; update driver set num_trips = num_trips + 1 where driver_id=2; commit;	start transaction; update driver set num_trips = num_trips + 1 where driver_id=2; commit;	This pair of queries do the same thing: increment value of one to the number of rides. The expected final value of num_trips should be incremented by 2. Let us say, the value of num_trips right now is 10

CONFLICT SERIALIZABLE SCHEDULING

	Transactional Query T1.	Transactional Query T2.	Remarks
Query:	start transaction; update driver set num_trips = num_trips + 1 where driver_id=2; commit;	start transaction; update driver set num_trips = num_trips + 1 where driver_id=2; commit;	This pair of queries do the same thing: increment value of one to the number of rides. The expected final value of num_trips should be incremented by 2. Let us say, the value of num_trips right now is 10
Schedule:	Read(A); //(where A is rating of driver id 2). A=A+1; Write(A); Commit;	Read(A) A=A+1 Write(A); Commit;	T1 reads A as 10 A=11 for T1 T1 writes A as 11. T1 finishes transaction T2 reads A as 11. Here, an WR conflict is caused, but as the scheduling is such that Read in T2 happens only after committing of T1, the read won't be buggy. A=12 for T2 T2 writes T1 as 12, which is a correct value.
Precedence graph:	 <pre>graph LR; T1((T1)) --> T2((T2))</pre>		
Conclusion:	As this scheduling didn't caused our database to get corrupted, this transaction scheduling is a good debugged scheduling. However, to ensure this Scheduling to work properly, we may use some locks such that T2 starts executing only after T1 gets committed. We can follow the same way of applying locks we did in non-conflict serializable scheduling..		

	NON-CONFLICT SERIALIZABLE SCHEDULING		
	Transactional Query T1.	Transactional Query T2.	Remarks
Query:	start transaction; update driver set num_trips = num_trips + 1 where driver_id=2; commit;	start transaction; update driver set num_trips = num_trips + 1 where driver_id=2; commit;	This pair of queries do the same thing: increment value of one to the number of rides. The expected final value of num_trips should be incremented by 2. Let us say, the value of num_trips right now is 10
Query schedule	Read(A); //(where A is rating of driver id 2).	Read(A);	Both the queries read A. At this point, none of the transactions have committed the value of A. Therefore, both T1 and T2 read A=10
	A=A+1;	A=A+1;	T1 has the value of A as 10+1=11 T2 also makes the value of A as 10+1=11.
	Write(A); Commit;	Write(A); Commit	T1 writes the value of A as 11 Here comes the condition of unrepeatable read(RW). T2 reads the value of A, which was being also used by T1, but T1 is not finished yet. T2 then overwrites the value of A as 11 also. This causes the condition of WW conflict. As both T1 and T2 had the value of A as 11, the final value of A written is 11. This is buggy, as the expected value of A was 12. This bug is caused by the RW conflict.
Precedence Graph			
Resolution:	Assign a 2PL lock for the memory of A. Whenever a transaction tries to read it, apply lock on A. When the transaction finished writing on it, release the lock, or wait for the commit.		
Edited transactions	start transaction; lock-X(A) where A is select num_trips from driver where driver_id==2. update driver set num_trips = num_trips + 1 where driver_id=2; commit;	start transaction; lock-X(A) where A is select num_trips from driver where driver_id==2. update driver set num_trips = num_trips + 1 where driver_id=2; commit;	In these transactions, an exclusive lock on A is used, so that each transaction can exclusively read and write on A, without any interference from other transactions. The lock is released automatically, when a transaction gets committed. As locking is atomic, the queries will avoid any deadlocks. The assumptions are same as above: num_trips=10
	Lock-X(A) //(Assumption: In the timestamp, the computer first starts executing T1.) Read(A) A=A+1 Write(A); Commit;	Lock-X(A) //Can't execute, has to wait for any previous locks on A to be released. Lock-X(A)//(executes, as commit releases the lock on A given by T1.) Read(A) A=A+1 Write(A) Commit;	Lock-X performed by T1 on A. T2 tries locking A, but has to wait for previous lock to get free. T1 reads A=10. A=11 in T1 A=11 written in memory As T1 commits, Lock-X(A) gets freed. T2 applied Lock-X on A. A=11 read by A. A=12 in T2 A=12 written in memory T2 releases lock.
Conclusion:	As we get the desired result after applying locking, that is, A=12, the previous bugs get resolved. This shows that locking indeed helps in preventing the usage of wrong data.		

	<p>-- deleting employee whose emp_id is 101 from all related tables.</p>	<p>-- updating membership date for a customer start transaction;</p>	<p>-- deleting the older feedbacks start transaction</p>	<p>-- increasing the salary of all types of employee start transaction;</p>
	<p>start transaction; delete from employee where emp_id=101; delete from employee_phone_no where emp_id=101; commit;</p>	<p>update membership set end_date='1996-01-22' where membership_id=(select membership_id from customer where customer_id=1); commit;</p>	<p>delete from customer_feedback where year(feedback_date)<=2020 and month(feedback_date)<=3; delete from driver_feedback where year(feedback_date)<=2020 and month(feedback_date)<=3; commit;</p>	<p>update employee set emp_salary=1.05*emp_salary where emp_role='Web Developer'; update employee set emp_salary=1.1*emp_salary where emp_role='HR'; update employee set emp_salary=1.1*emp_salary where emp_role='Manager'; update employee set emp_salary=1.05*emp_salary where emp_role='Customer Support'; commit;</p>
	<p>Read(A)//(A is employee table) delete(emp_id=101)//(delete entry of id of 101 from A) write(A) Read(B)//(B is table of phone no.s) delete(emp_id=101)//(delete all entries where emp_id==101) write(B)</p>	<p>Read(A)//(where A is membership of selected customer) update(A)//(A's date is updated) write(A)</p>	<p>Read(A)//customer_feedback table delete(A)//(delete all entries with given parameters) write(A)</p>	<p>Read(A)//(employee table with) update(A)//(update the salary of corresponding role, with corresponding value) write(A)</p>