QuickCart : An Online Retail Store

Aarzoo(2022008), Shobhit Raj(2022482), Sidhartha Garg(2022499), Vanshika Pal (2022560)

April 20, 2024

Non-Conflicting Database Transactions

Assuming the application is hosted on a public server, multiple users can access and use it simultaneously. Below cases demonstrate scenarios where the transactions are non-conflicting.

Transaction Pair 1 (Delivering Orders)

Consider the case of two delivery agents delivering their respective orders simultaneously.
The following table of transactions sums up the reads and writes to the database needed to complete the required actions.
Here, S_1 and S_2 represent the status of orders O_1 and O_2, respectively.
W_1 and W_2 represent the wallet of agents A_1 and A_2,respectively.
Similarly, D_1 and D_2 represent the status of agents A_1 and A_2,respectively.

Transaction-1 (T_1)	**Transaction-2** (T_2)
$S_1 :=$ *Delivered*	$S_2 :=$ *Delivered*
Write(Order 1 status)	Write(Order 2 status)
Read(Agent 1 Wallet)	Read(Agent 2 Wallet)
$W_1 := W_1 + c_1$	$W_2 := W_2 + c_2$
Write(Agent 1 Wallet)	Write(Agent 2 Wallet)
$D_1 :=$ *Available*	$D_2 :=$ *Available*
Write(Agent 1 status)	Write(Agent 2 status)
Commit	Commit

Transaction Pair 2 (Browsing and Adding to Cart)

Consider the case involving browsing of products by 2 different users and adding them to their respective carts without making a purchase.
The following table of transactions sums up the reads and writes to the database needed to complete the required actions.
Here , Q_1 represents quantity of product 1 added to cart C_1, Q_2 represents quantity of product 1 added to cart C_2.
C_1 and C_2 represents the cart of the respective users.

Transaction-1 (T_1)	**Transaction-2** (T_2)
Read(product 1 details)	Read(product 1 details)
$C_1 := C_1 + Q_1$	$C_2 := C_2 + Q_2$
Write(Add product 1 to cart 1)	Write(Add product 1 to cart 2)
Commit	Commit

Transaction Pair 3 (Giving Order Review)

Consider the case of two customers simultaneously giving order reviews of their respective orders.
The following table of transactions sums up the reads and writes to the database needed to complete the required actions
Here, R_1 and R_2 represent the order reviews of orders O_1 and O_2, respectively.

Transaction-1 (T_1)	**Transaction-2** (T_2)
$R_1 := review$	$R_2 := review$
WRITE(ORDER 1 REVIEW)	WRITE(ORDER 2 REVIEW)
COMMIT	COMMIT

Transaction Pair 4 (Withdraw Money)

Consider the case of two delivery agents withdrawing money from their respective wallets.
The following table of transactions sums up the reads and writes to the database needed to complete the required actions.
W_1 and W_2 represent the wallet of agents A_1 and A_2,respectively.
Similarly, T_1 and T_2 represent the amounts they are withdrawing from their respective wallets.

Transaction-1 (T_1)	**Transaction-2** (T_2)
READ(AGENT 1 WALLET)	READ(AGENT 2 WALLET)
$W_1 := W_1 - T_1$	$W_2 := W_2 - T_2$
WRITE(AGENT 1 WALLET)	WRITE(AGENT 2 WALLET)
COMMIT	COMMIT

Any scheduling or order of execution of these pair of transactional operations will not result in conflict as the read and write operations are performed on different rows.

Conflicting Database Transactions

Assuming the application is hosted on a public server, multiple users can access and use it simultaneously. Below cases demonstrate scenarios where the transactions can be conflicting.

Transaction Pair 1

A conflicting transaction where two users attempt to buy the same product, at the same time. This represents a situation where the transactions conflict due to a resource constraint (product stock)

Transaction-1 (T_1)	**Transaction-2 (T_2)**
Read(User 1 balance)	Read(User 2 balance)
Read(stock)	Read(stock)
Write(decrement stock)	Write(decrement stock)
Write(User 1 new balance)	Write(User 2 new balance)
Commit	Commit

Conflict Serializable

Transaction-1 (T_1)	**Transaction-2 (T_2)**
Read(User 1 balance)	
	Read(User 2 balance)
Read(stock)	
Write(decrement stock)	
	Read(stock)
Write(User 1 new balance)	
	Write(decrement stock)
Commit	
	Write(User 2 new balance)
	Commit

**Here, we can see a 'dirty read' occurs if transaction1 rolls back on some failure and transaction2 commits.

Transaction Pair 2

A conflicting transaction between an admin deleting a product and a customer checking out with that product.

Admin's Transaction (T_{admin})	**Customer's Transaction ($T_{customer}$)**
Read(product1 to be deleted)	Read(product1 stock)
Write(delete product1)	Write(decrement product1 stock)
Commit	Write(payment details)
	Commit

Conflict Serializable

Admin's Transaction (T_{admin})	**Customer's Transaction ($T_{customer}$)**
Read(product1 to be deleted)	
	Read(product1 stock)
Write(delete PRODUCT1)	
	Write(decrement PRODUCT1 stock)
	Write(payment details)
Commit	
	Commit

**Here, we can see a conflict where a product does not exist but the customer checks out the product

Transaction Pair 3

Conflicting transactions may arise when two customers tip the same delivery agent for their past orders. Ideally, after both transactions have executed, the delivery agent's wallet balance should be the sum of both tips plus the old balance (T1 + T2 + old balance). However, due to the scheduling of operations, the balance might be less, specifically the sum of the second tip and the old balance (T2 + old balance).

Transaction-1 (T_1)	**Transaction-2** (T_2)
READ(deliveryAgentWallet balance)	READ(deliveryAgentWallet balance)
A1 := balance + $T1$	A2 := balance + $T2$
WRITE(deliveryAgentWallet A1)	WRITE(deliveryAgentWallet A2)
COMMIT	COMMIT

Confict Serializable

Transaction-1 (T_1)	**Transaction-2** (T_2)
READ(deliveryAgentWallet balance)	READ(deliveryAgentWallet balance)
A1 := balance + $T1$	A2 := balance + $T2$
WRITE(deliveryAgentWallet A1)	
	WRITE(deliveryAgentWallet A2)
COMMIT	COMMIT

**Here, we can see a conflict where both customers are trying to write to a particular delivery agent's wallet