

Isolation

by Sophia



WHAT'S COVERED

This lesson explores the isolation property in a transaction and how it affects the database, in two parts. Specifically, this lesson will cover:

1. [Introduction](#)
2. [Isolation Example](#)

1. Introduction

The **isolation** property in a transaction ensures that if there are multiple transactions run at the same time as one another, they do not leave the database in an inconsistent state. The transactions themselves should not interfere with one another, and each transaction should be run independently. Any changes that statements in a transaction are making will be visible only to that transaction, and any other concurrent transaction will not see the results of the changes until the transaction is complete and the data has been committed. This also ensures that transactions that run concurrently have the same results as if they were run sequentially.

In addition, the isolation property ensures that the data that is used in a transaction cannot be used in another transaction until the original transaction is complete with it. Most databases, including PostgreSQL, will use locking to maintain transactional isolation.



TERM TO KNOW

Isolation

The quality of ensuring that concurrent transactions are isolated from each other, preventing interference with one another's operations and maintaining data integrity.

2. Isolation Example

Let us consider our banking example yet again.

⇒ **EXAMPLE** Jennifer would like to make a \$100 payment to Randall through an account transfer. This transaction is a balance transfer between two accounts at two different branches of the same bank. Here's what the transaction looks like:

1. \$100 would be deducted from Jennifer's account at branch 10.
2. Branch 10's balance would be decreased by \$100.
3. Branch 50's account would be increased by \$100.
4. Randall's account at branch 50 would be increased by \$100.

Let us look at the transaction in SQL:

```
BEGIN;
UPDATE customer_account
SET balance = balance - 100
WHERE account_id = 10;
UPDATE branch_account
SET balance = balance - 100
WHERE branch_id = (SELECT branch_id FROM customer_account where account_id = 10);
UPDATE branch_account
SET balance = balance + 100
WHERE branch_id = (SELECT branch_id FROM customer_account where account_id = 50);
UPDATE customer_account
SET balance = balance +100
WHERE account_id = 50;
COMMIT;
```

IN CONTEXT

Let us say that both Jennifer's and Randall's account balances started at \$1000. If Jennifer was attempting to start this transaction and Randall concurrently was trying to check his account balance, Randall should not see any updates to his account until the changes are made and the transaction has been committed. If Randall queries for his customer_account balance, it would be at \$1000 until the entire transaction from Jennifer executes successfully and commits the data to the database. In certain databases, if Randall tried to query his account balance, no result would be provided until Jennifer's transaction was completed.

Isolation is increasingly important as more concurrent transactions access the same data. For example, imagine a situation where Randall is receiving two different account transfers from two different individuals at the same time.

Imagine Randall is receiving \$100 from Jennifer and \$50 from Paul. If there were no isolation in place, Jennifer's transaction could check Randall's balance at \$1000. At the same time, Paul's transaction could check Randall's balance at \$1000. Jennifer's transaction would add \$100 to the \$1000 and save

the results. Paul's transaction would add \$50 to the \$1000 and save the results. The final result is that Randall's account balance is \$1050 instead of \$1150. This could be the case if Jennifer's and Paul's transactions are both reading Randall's balance at the same time. Jennifer's transaction started and updated the balance, but Paul's transaction was also completed and saved over Jennifer's transaction.

This is why isolation is important, as Jennifer's transaction would prevent Paul's transaction from reading the value. Even if the value is read, Paul's transaction would not be able to complete due to the inconsistency in the database state.



SUMMARY

In this lesson, in the **introduction**, you learned that concurrent transactions should be isolated from one another in accordance with the ACID model (Atomicity, Consistency, Isolation, Durability) to ensure data integrity. A multiuser database environment may lead to conflicts due to simultaneous transactions, such as "lost updates" or "dirty reads," when one transaction adversely affects another. The isolation mechanism ensures that each transaction appears to run in isolation, oblivious to the presence of other concurrent transactions. Different mechanisms enable isolation, such as locking, which prevents multiple transactions from modifying the same data at the same time.

In the **isolation example**, you considered two transactions modifying the same record simultaneously. The isolation process ensures that only one transaction can modify a record at a time, while all other transactions are put on hold until the first transaction is complete. This prevents data corruption or discrepancies from occurring due to simultaneous modifications.

Source: THIS TUTORIAL WAS AUTHORED BY DR. VINCENT TRAN, PHD (2020) AND FAITHE WEMPEN (2024) FOR SOPHIA LEARNING. PLEASE SEE OUR [TERMS OF USE](#).



TERMS TO KNOW

Isolation

The quality of ensuring that concurrent transactions are isolated from each other, preventing interference with one another's operations and maintaining data integrity.