



CSE 512 Distributed Database Systems

Project Part - 4

ACID Properties:

Atomicity

All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are. For example, in our application that transfers funds from one account to another, the atomicity property ensures that, if a debit is made successfully from one account, the corresponding credit is made to the other account.

Consistency

Data is in a consistent state when a transaction starts and when it ends. For example, in an application that transfers funds from one account to another, the consistency property ensures that the total value of funds in both accounts is the same at the start and end of each transaction.

Isolation

The intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be serialized. For example, in an application that transfers funds from one account to another, the isolation property ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither.

Durability

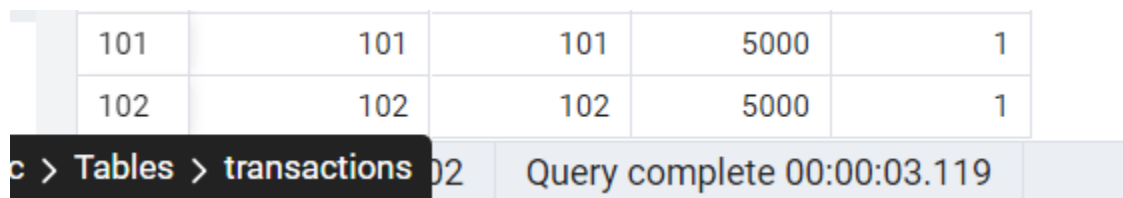
After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure. For example, in an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed.

The `psycpg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT` setting in PostgreSQL specifies that each SQL statement should be treated as a transaction and automatically committed to the database as soon as it is executed. This ensures that any changes are immediately committed.

We have handled transactions between two users where the amount gets deducted from the sender and gets added to the receiver. The total amount of the sender + receiver remains the same before and after the transaction.

To distinguish between the success and failure of a transaction, we have three statuses. The amount gets debited or credited only when the transaction status is "SUCCESS".

This helps us in achieving ACID properties. The replica table helps us in making the system more durable. The amounts transferred are either transferred to one account or not proving isolation. Total value of the funds between the two accounts before and after the transaction are the same ensuring consistency. A failure in insertion or command, cause the transaction to be aborted ensuring atomicity.

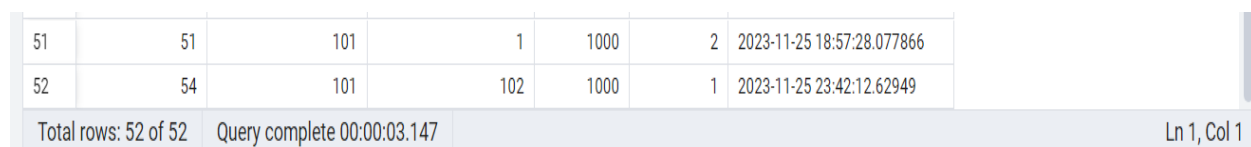


101	101	101	5000	1
102	102	102	5000	1

c > Tables > transactions 02 Query complete 00:00:03.119

Fig 1. Accounts table upon insertion of two new rows

Now let us make a transaction of 1000 from 101 to 102 with status as SUCCESS



51	51	101	1	1000	2	2023-11-25 18:57:28.077866
52	54	101	102	1000	1	2023-11-25 23:42:12.62949

Total rows: 52 of 52 Query complete 00:00:03.147 Ln 1, Col 1

Fig 2. Creating a transaction between users 101 and 102

100	100	100	4769	1
101	101	101	4000	1
102	102	102	6000	1
Total rows: 102 of 102			Query complete 00:00:01.557	

Fig 3. Accounts table after the transaction

Concurrent Transactions

We simulate concurrent transactions by creating and starting multiple threads to perform concurrent queries. In this case, we start with five threads, each representing a database query.

It takes a `query_id` parameter, and depending on the value of `query_id`, it either performs a simulated UPDATE operation (for `query_id == 1`) or a simulated SELECT operation (for other `query_id` values).

In summary, the code simulates concurrent database operations using threads. Each thread performs either a simulated UPDATE or SELECT operation on the same selected database record, demonstrating how multiple threads can interact with a database concurrently.

```
PS C:\Users\aramak21> & C:/Python311/python.exe c:/Users/aramak21/Downloads/dds_proj.py
established connection to Postgres!!
A database named trn2200 already exists
Thread 3: Simulated SELECT logic. Result: [(101, 'UserX1', 'user1@example.com', 987654301, 1)]
Thread 2: Simulated SELECT logic. Result: [(101, 'UserX1', 'user1@example.com', 987654301, 1)]
Thread 1: Simulated UPDATE logic. Record updated.
Thread 5: Simulated SELECT logic. Result: [(101, 'UserX1', 'user1@example.com', 987664301, 1)]
Thread 4: Simulated SELECT logic. Result: [(101, 'UserX1', 'user1@example.com', 987664301, 1)]
All queries completed.
PS C:\Users\aramak21>
```

Fig 4. Concurrent queries on the database

Thread 1 performs the update query. As we can see, the queries following it have a different phone number than the queries above it. Thus our system is capable of handling queries concurrently while ensuring that the results are consistent.