Paseo Posse – Hyperlocal Delivery System

ACID is an acronym that represents a set of properties that guarantee the reliability of database transactions. These properties are crucial for maintaining the consistency and integrity of data in a distributed database environment:

Atomicity:

Definition: Atomicity ensures that a transaction is treated as a single, indivisible unit. Either all of its operations are executed, or none are.

Enforcement in our Distributed Databases: This property is maintained by using distributed transaction management protocols. When a distributed transaction involves multiple nodes, a two-phase commit protocol is employed to ensure atomicity.

Consistency:

Definition: Consistency ensures that a transaction brings the database from one valid state to another. It prevents the database from being left in an intermediate, invalid state.

Enforcement in our Distributed Databases: Consistency is often enforced through the careful design of the database schema, relationships, and constraints. SQL features such as FOR UPDATE help in maintaining consistency during concurrent transactions.

Isolation:

Definition: Isolation ensures that the execution of one transaction is isolated from the execution of other transactions. Transactions appear as if they are executed serially, even when multiple transactions are executed concurrently.

Enforcement in our Distributed Databases: Isolation levels, such as READ COMMITTED and REPEATABLE READ, define the level of isolation between transactions. Techniques like skip locks and selective indexing contribute to maintaining isolation and preventing interference between transactions.

Durability:

Definition: Durability guarantees that once a transaction is committed, its changes persist even in the event of a system failure.

Enforcement in our Distributed Databases: write-ahead logging and replication. In a distributed environment, replication across multiple nodes enhances durability by ensuring data redundancy.

Concurrency Control:

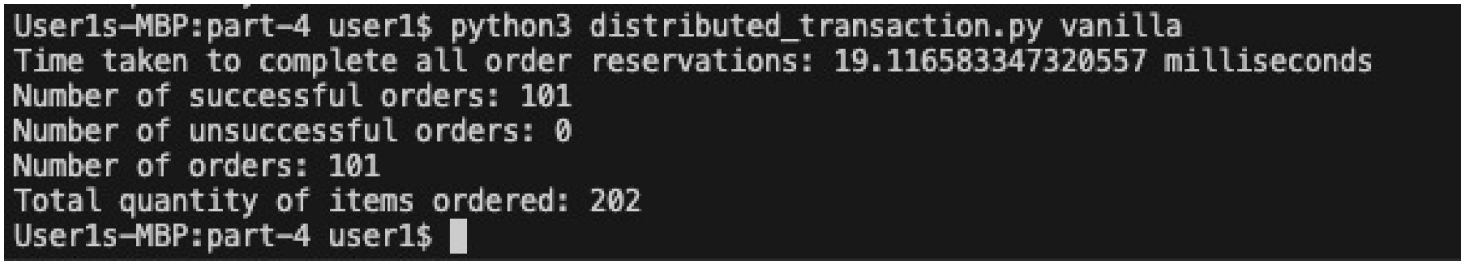
Concurrency control is the mechanism that ensures that multiple transactions can be executed concurrently without leading to inconsistencies in the database. Key aspects of concurrency control include:

Locking:

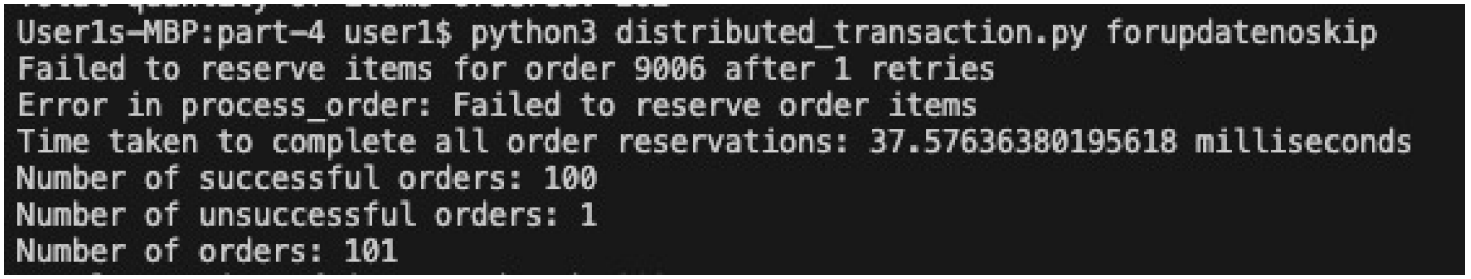
Transactions use locks to control access to shared resources. For example, the FOR UPDATE clause in SQL queries acquires locks on rows, preventing other transactions from modifying the same rows simultaneously.

For instance, consider we have 200 medicines and 101 customers. Each customer orders 2 medicines. So, 100 transactions will be successful, and 1 customer will not get any medicines.

Below is a screenshot of a case where there is **no FOR UPDATE clause**, so a single resource can be called by multiple processes, and 202 medicines are ordered, which is incorrect.

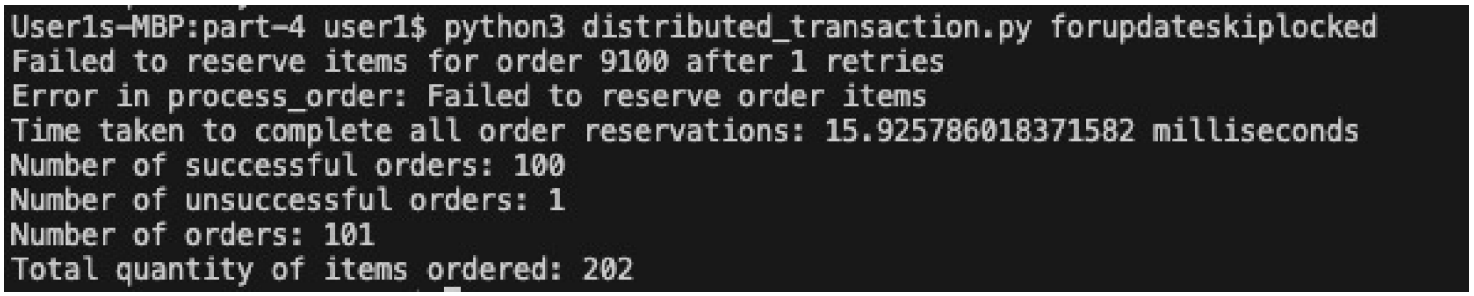


Below is a screenshot of a case where there is **a FOR UPDATE clause**, so a single resource cannot be called by multiple processes, and only 100 transactions are successful, and 1 transaction fails. But at a **time only one process can access resources and others have to wait**.



Below is a screenshot of a case where there **is a FOR UPDATE clause with SKIP LOCKED**, so a single resource cannot be called by multiple processes, and only 100 transactions are successful, and 1 transaction fails. Here at a time **multiple processes can access resources and no one has to wait**.

Thus, we see the time taken for this is way lower. There 57.6% reduction in time taken than earlier queries.



Isolation Levels:

Isolation levels define the visibility of changes made by one transaction to other concurrent transactions. Levels such as READ COMMITTED and REPEATABLE READ offer different trade-offs between consistency and performance.

Optimistic Concurrency Control:

Rather than locking resources, optimistic concurrency control allows transactions to proceed without locks, and conflicts are detected at the time of commit. This approach is suitable for scenarios with low contention.

Distributed Coordination for Transactions:

Two-Phase Commit (2PC):

2PC is a protocol that we have used to ensure the atomicity of distributed transactions. It involves a coordinator and participants. In the first phase, the coordinator asks participants if they are ready to commit. In the second phase, based on participants' responses, the coordinator decides whether to commit or abort.