Hotel Management System

Transaction scenario 1:

Reserving a room or check-in process:

When a receptionist creates a reservation, the system must:

1. Insert reservation record
2. Update room status to ‘reserved’
3. Create invoice
4. Process initial payment

All these operations must succeed together or fail together.

Atomicity (All or Nothing)

Atomicity ensures that the booking process is integral.

Example:

- If invoice creation fails
- Or payment processing fails
- Or room update fails

Then:

- Reservation record is removed

- Room status is not changed
- No partial data remains

Without atomicity:

We might have a reserved room but no invoice.Or payment recorded without reservation. This would corrupt the system.

Consistency (Valid State Before and After)

Consistency ensures database rules are never violated.

In this scenario consistency is maintained by:

- Foreign Keys:
    - reservation must reference valid guest
    - invoice must reference valid reservation
- Check constraint:
    - check_out_date > check_in_date
- Enum validation:
    - room status must be one of (available, reserved, occupied, maintenance)
- NOT NULL constraints
- Payment amount must be > 0
- Before transaction:
  Room status = available
- After transaction:
  Room status = reserved

Invoice created

Payment recorded

Database remains in a valid state.

Isolation (Handling Concurrent Bookings)

Imagine two receptionists try to book the same room at the same time.

Without isolation:
Both could see room as available and double-book it.

With proper isolation:

- When first transaction updates the room
- Database locks that row
- Second transaction must wait

This prevents:

- Dirty reads
- Lost updates
- Double booking

We use:

- Row-level locking
- Indexed search on room_id
- Transaction boundaries

Durability (Permanent Once Committed)

After Commit:

- Reservation is permanently stored
- Invoice exists
- Payment recorded
- Room marked reserved

Even if server crashes or power failure occurs.

Transaction scenario 2:

Check-out process:

During checkout the system must:

1. Add all service charges to invoice
2. Process remaining payment
3. Update invoice status to 'paid'
4. Update room status to 'available'
5. Mark reservation as 'checked_out'
6. Award loyalty points

All operations must succeed together.

Atomicity

If loyalty points update fails, entire checkout must fail.

If payment fails, room should not become available.

Without atomicity:

- Guest might leave
- Room marked available
- But payment incomplete

Atomicity ensures financial correctness.

Consistency

Consistency ensures:

- Invoice total = subtotal + tax − discount
- Payment_status changes to 'paid' only when fully paid
- Room status becomes available only after checkout
- Loyalty points = floor(total_amount / 10)

Example:
If total bill = $450
Points awarded = 450 / 10 = 45 points

All constraints ensure business rules are preserved.

Isolation

During checkout:

- No other staff should modify the same invoice
- No other reservation should claim the same room

Isolation prevents:

- Dirty reads (reading uncommitted payment)
- Non-repeatable reads
- Lost updates

We use:

- Row-level locks on:
    - reservations
    - invoices
    - rooms
    - guests
- Indexed queries for faster locking

Durability

After checkout commit:

- Payment permanently recorded
- Invoice marked paid
- Guest loyalty updated
- Room available

Even in case of system crash, transaction log ensures recovery.

Concurrency Control Strategy:

Since many staff members use the system at the same time, problems can occur. For example, two receptionists might try to book the same room for different guests. To prevent errors like these, we need a system to manage everyone working at once.

All-or-Nothing Transactions:
Important tasks (like booking a room and making payment) are treated as one complete action. Either everything succeeds, or nothing happens. This prevents problems like charging money without confirming the booking.

Locking:
When someone edits or books something, only that specific item is locked. For example, if one room is being booked, only that room is blocked for others, the rest of the system still works normally.

Keeping Relationships Valid:
The system checks that all related data makes sense. For example, it won't allow a reservation for a guest who isn't in the system. This keeps the data correct and reliable.

Faster Searching:
The database is optimized to find information quickly. Because tasks finish faster, locks are released sooner, and users don't have to wait long to use the system.

Isolation Level Justification:

Isolation controls when one person's work becomes visible to others using the system at the same time.

If isolation isn't set correctly, problems like these can happen:

- Two staff members think the same room is free.
- Someone sees an unpaid bill as paid before it's confirmed.
- Daily financial reports show wrong numbers.
- Guest loyalty points are calculated incorrectly.

Repeatable read:

No Dirty Reads:
You cannot see changes that are not finished yet.
Example: If a payment is still being processed, other users cannot see it until it is completed. This keeps data accurate.

No Non-Repeatable Reads:
If you read the same data twice during one task, it will not change in between.
Example: When booking a room, availability checked at the start will stay the same until the booking is done, so you don’t accidentally reserve a room that someone else took.

No Lost Updates:
If two users try to change the same data at the same time, the system handles them one by one.
Example: One person’s update will not overwrite another person’s update accidentally.