ACID Documentation

Transaction Scenario 1: Carpool Seat Booking

A passenger books seats in a carpool offer. Three operations must succeed as one unit or all roll back.

BEGIN;

Step 1: Lock the offer row to prevent concurrent overbooking

SELECT available_seats FROM carpool_offers
WHERE carpool_id = 7 FOR UPDATE;

Step 2: Book the seat (trigger checks & decrements available_seats)

INSERT INTO carpool_bookings (carpool_id, passenger_id, seats_booked, status)
VALUES (7, 42, 2, 'confirmed');

Step 3: Record payment

INSERT INTO payments (user_id, booking_type, booking_id, amount, payment_method, status)
VALUES (42, 'carpool', currval('carpool_bookings_booking_id_seq'), 300.00, 'card', 'completed');
COMMIT;

Note : If any step fails, vehicle stays available and no booking is recorded

ACID Property	How It Is Satisfied
Atomicity	All three steps (seat lock, carpool_bookings INSERT, payments INSERT) are inside a single BEGIN...COMMIT block. If the payment INSERT fails (e.g., constraint violation), the DBMS rolls back the carpool_bookings row
Consistency	The BEFORE INSERT trigger trg_check_carpool_seats aborts the transaction if available_seats < seats_booked. The AFTER INSERT trigger trg_carpool_seats_update decrements available_seats and sets status = 'full' when seats reach zero. The CHECK (seats_booked > 0) and UNIQUE (carpool_id, passenger_id) constraints prevent invalid or duplicate bookings.
Isolation	Isolation level: READ COMMITTED. The SELECT FOR UPDATE on carpool_offers locks the row, serializing concurrent bookings. A second transaction attempting the same carpool_id waits until the first commits or rolls back, preventing the lost-update anomaly where two passengers simultaneously see seats = 1 and both succeed.
Durability	Once COMMIT executes, the DBMS guarantees all changes are written to durable storage before returning success. The booking and payment records will survive any subsequent crash or power failure.

Transaction Scenario 2: Rental Booking with Vehicle Status Update

A customer creates an active rental. The vehicle must be marked unavailable and payment recorded atomically.

BEGIN;

Step 1: Lock vehicle row to prevent double-booking

SELECT is_available FROM vehicles WHERE vehicle_id = 3 FOR UPDATE;

Step 2: Create the rental (trigger sets vehicles.is_available = FALSE)

INSERT INTO rental_bookings (customer_id, vehicle_id, start_date, end_date, total_amount, status)
VALUES (15, 3, '2026-03-01', '2026-03-05', 2000.00, 'active');

Step 3: Record payment

INSERT INTO payments (user_id, booking_type, booking_id, amount, payment_method, status)
VALUES (15, 'rental', currval('rental_bookings_rental_id_seq'), 2000.00, 'card', 'completed');

COMMIT;

Note : If any step fails, vehicle stays available and no rental is recorded

ACID Property	How It Is Satisfied
Atomicity	All three operations (rental INSERT, vehicle UPDATE via trigger, payment INSERT) run inside one BEGIN...COMMIT. If the payment INSERT violates any constraint, the DBMS rolls back the rental row and the trigger's vehicle flag change together. The vehicle is never left marked unavailable without a matching booking.
Consistency	The BEFORE INSERT trigger trg_check_rental_start_date aborts if start_date is in the past. CHECK (end_date >= start_date) prevents invalid date ranges. CHECK (total_amount >= 0) prevents negative charges. The AFTER INSERT/UPDATE trigger trg_rental_vehicle_availability keeps vehicles.is_available in sync with rental status at all times.
Isolation	Isolation level: READ COMMITTED. The SELECT FOR UPDATE on vehicles locks that row for the duration of the transaction. A second concurrent session trying to book the same vehicle is blocked until the first commits or rolls back, preventing the double-booking anomaly.
Durability	Once COMMIT executes, the DBMS guarantees all three changes (rental row, vehicle flag, payment row) are persisted in storage.

Concurrency Strategy & Isolation Level Justification

Concern	Decision & Rationale
Isolation level chosen	READ COMMITTED for all booking transactions. It prevents dirty reads and is sufficient because write conflicts on critical rows are resolved by separate SELECT FOR UPDATE locks rather than a higher isolation level. This keeps good concurrency for read-heavy views while keeping write transactions safe.
Write concurrency (booking rows)	Pessimistic locking via SELECT FOR UPDATE on the shared resource row (carpool_offers or vehicles). The first transaction holds the lock, concurrent transactions queue behind it, eliminating lost-update and double-booking anomalies and errors.
Read concurrency (reporting views)	No explicit locking needed. Under READ COMMITTED, readers always see a consistent snapshot of committed data and never block writers.
Why not SERIALIZABLE?	SERIALIZABLE would be unnecessary as we can do our work just fine with READ COMMITTED + SELECT FOR UPDATE combo. In SERIALIZABLE, many transactions fail and users must retry moreover using it will be slower and will effect user experience.

Extra transaction scenarios (just for our understanding):

Transaction : create rental booking with payment

Scenario:
The customer books a vehicle rental and makes a payment.

The transaction has 4 steps,

- check vehicle availability
- Create booking
- Record payment
- Update vehicle status

Atomicity:
if any of the above steps fail, all changes are rolled back. As all the steps are inside Begin and COMMIT block, if any fails the whole transaction is rolled back.

Consistency:

The transaction maintains all integrity constraints and moves database from one valid state to another.

Constraints:

- Rental_id in rental_payments must exist in rental_bookings
- CHECK that the amount user paid >= 0
- CHECK end_date > start_date (cannot end rental before it starts)
- Customer_id is not null

State validation:
Lets say before we had 30 booking that costed 100,000 and the total payments received are 100,000. After a new booking, bookings = 31 their amount = 110,000 and total payments amount also increases to 110,000. (there will be profits ik, but just a simple example to show consistent states)

Isolation:
Prevents dirty reads and ensures concurrency.
READ COMMITTED and SELECT FOR UPDATE are used. READ COMMITTED is also the default isolation level in postgresql.
SELECT FOR UPDATE acquires an exclusive lock on a row making other transaction wait until this one completes on this row.

Durability:

Once COMMIT is successful, the booking and payment are permanent and will survive any system failure. In PostgreSQL, WAL or in simpler words log files are used to ensure this.

Transaction : Driver Accepts Ride request

Multiple drivers see the same pending ride. When multiple drivers tap "Accept" simultaneously, only one should succeed. Others should immediately know the ride is taken.

Atomicity:
The driver assignment either completely fails or completely happens.

Consistency:
Each ride has zero or one driver at all times. CHECKS/ FKs NOT NULLS are used so that db cannot enter an invalid state.

Isolation:
Lets say multiple drivers are shown a pending ride, without locking, all drivers can be able to accept the ride and the customer will be shown multiple drivers breaking the whole logic.

Durability:
Once committed, the transaction survives.