Food Delivery System

ACID Documentation

Transaction Scenarios | Concurrency Strategy | Isolation Levels

Overview

This document maps four key database transactions in the Food Delivery System to ACID properties. Each scenario covers the transaction steps, ACID mapping, concurrency control strategy, and isolation level justification. A comparative isolation level reference and summary are included at the end.

Transaction Scenario 1: Placing an Order

Steps

1. INSERT INTO Order — create a new order record with customer, restaurant, and address references.
2. INSERT INTO OrderItem (x N) — insert one row per item selected by the customer.
3. Trigger (trg_order_total_after_insert) — auto-recalculates and writes Order.total_amount.
4. INSERT INTO Payment — create a payment record if the customer chose a prepaid method.

ACID Mapping

	Property	Description
A	**Atomicity**	All four steps succeed together or the entire transaction rolls back. If the Payment INSERT fails, the Order and OrderItem rows are also rolled back — no orphaned records.
C	**Consistency**	FK constraints guarantee valid customer_id, restaurant_id, and address_id. CHECK (price > 0) and CHECK (quantity > 0) prevent invalid line items. The database never holds a dangling order.
I	**Isolation**	SELECT ... FOR UPDATE on MenuItem rows prevents a concurrent transaction from marking an item unavailable mid-order. Other transactions see either the fully committed order or nothing.
D	**Durability**	Once COMMIT executes, the write-ahead log (WAL) ensures order data survives a server crash before pages are flushed to disk.

Concurrency Strategy

Strategy	Pessimistic locking — SELECT ... FOR UPDATE is issued on each MenuItem row before checking availability. This prevents two concurrent customers from ordering the last unit of an item simultaneously.
Isolation Level	READ COMMITTED
Justification	Prevents dirty reads (e.g., reading an uncommitted payment status). Suitable for high-concurrency order placement without serializing all reads. SERIALIZABLE would cause excessive lock contention on a busy menu without meaningful correctness benefit.

Transaction Scenario 2: Completing a Delivery

Steps

1. UPDATE Order SET status = 'Out for Delivery', rider_id = ? WHERE order_id = ? — rider is assigned and status advances.
2. UPDATE Order SET status = 'Delivered' WHERE order_id = ? — rider marks delivery complete.
3. INSERT INTO Review (...) — optional: customer is prompted to submit a review after delivery confirmation.

ACID Mapping

	Property	Description
A	**Atomicity**	The status update and any related audit log insert must both succeed. If writing the log fails, the status change is rolled back — the order does not show Delivered without a corresponding record.
C	**Consistency**	Trigger (trg_prevent_invalid_status) validates transition order — cannot jump from Pending to Delivered, skipping Preparing. ENUM CHECK prevents invalid status values entirely.
I	**Isolation**	A row-level lock on the Order record ensures no two riders can simultaneously update the same order. An updated_at timestamp is checked before each UPDATE to detect stale reads.
D	**Durability**	Delivery confirmation is critical — WAL guarantees the Delivered status is persisted to disk and survives crashes, ensuring customers and riders see accurate final state.

Concurrency Strategy

Strategy	Optimistic concurrency — before issuing UPDATE, the application reads the current updated_at timestamp. The UPDATE includes WHERE updated_at = <read_value>. If another process modified the row in between, affected row count = 0 and the application retries. Avoids holding locks during network round-trips.
Isolation Level	READ COMMITTED
Justification	Allows reading the latest committed delivery status without blocking concurrent reads by the customer or restaurant dashboard. REPEATABLE READ is unnecessary because each step reads the row once and re-reads within the same transaction are not required.

Transaction Scenario 3: Admin Adding a Restaurant

Steps

1. INSERT INTO Restaurant (...) — admin creates a new restaurant record with name, location, phone, and unique email.
2. INSERT INTO Category (...) x N — admin inserts one or more menu categories for the restaurant.
3. INSERT INTO MenuItem (...) x N — admin populates menu items with price and availability flags.
4. INSERT INTO Admin_Manages (...) — relationship record linking the admin to the restaurant is created with managed_at timestamp.

ACID Mapping

	Property	Description
A	**Atomicity**	If any step fails (e.g., duplicate restaurant email), all prior inserts in the transaction roll back. The system never has a Restaurant with no categories, or orphaned MenuItems.
C	**Consistency**	UNIQUE constraint on Restaurant.email prevents duplicates. FK on Category.restaurant_id and MenuItem.category_id enforce referential integrity. CHECK (price > 0) validates all menu item prices.
I	**Isolation**	A shared intent lock on the Restaurant table prevents another admin from inserting a duplicate restaurant concurrently. Row-level locks are used for UPDATE operations on existing restaurants.
D	**Durability**	Once committed, the new restaurant and its full menu are immediately durable and visible to customers browsing the platform, backed by WAL persistence.

Concurrency Strategy

Strategy	Pessimistic locking — a table-level intent lock is acquired when inserting a new restaurant to prevent duplicate email violations from two concurrent admin sessions. Row-level locks are used for existing restaurant edits.
Isolation Level	READ COMMITTED
Justification	Admin operations are low-frequency compared to customer reads, so READ COMMITTED provides sufficient protection. It prevents an admin from reading another admin's uncommitted restaurant draft while allowing customers to concurrently browse already-committed data without blocking.

Transaction Scenario 4: Processing a Payment Refund

Steps

1. SELECT * FROM Payment WHERE order_id = ? FOR UPDATE — lock the payment row to prevent concurrent refund attempts on the same order.
2. UPDATE Order SET status = 'Cancelled' WHERE order_id = ? — cancel the associated order before processing the refund.
3. UPDATE Payment SET status = 'Refunded' WHERE payment_id = ? — mark payment as refunded.
4. INSERT INTO RefundLog (...) — create an audit record with timestamp and admin reference.

ACID Mapping

	Property	Description
A	**Atomicity**	Order cancellation and payment refund must be a single atomic unit. A partial failure (e.g., payment status updated but order not cancelled) would leave the system in an inconsistent billing state — full rollback required.

	Property	Description
C	**Consistency**	Payment.status ENUM only allows Paid, Failed, or Refunded. A trigger enforces that a refund can only be issued if current status is Paid and Order.status is not already Delivered.
I	**Isolation**	SELECT ... FOR UPDATE on the Payment row prevents two support agents from issuing a double refund concurrently. No other transaction can read or modify this row until the refund commits.
D	**Durability**	Refund records are financial data — WAL and synchronous commit ensure the Refunded status and RefundLog entry are fully persisted before the transaction is acknowledged to the caller.

Concurrency Strategy

Strategy	Pessimistic locking — SELECT ... FOR UPDATE is used at the start to immediately lock the Payment row. Essential for financial transactions where optimistic retry would risk a double-refund if two agents act simultaneously.
Isolation Level	REPEATABLE READ
Justification	Elevated to REPEATABLE READ for financial correctness. The transaction reads Payment.status at the start and must see the same value throughout. A non-repeatable read — another transaction changing status between the SELECT and UPDATE — could cause an erroneous double refund. The stronger guarantee justifies the slight reduction in concurrency.

Isolation Level Reference

Comparison of all four SQL standard isolation levels. Green rows indicate the levels chosen for this system.

Isolation Level	Dirty Read	Non-repeatable Read	Phantom Read	Used In
`READ UNCOMMITTED`	Possible	Possible	Possible	Rarely used
`READ COMMITTED`	Prevented	Possible	Possible	Scenarios 1, 2, 3
`REPEATABLE READ`	Prevented	Prevented	Possible	Scenario 4
`SERIALIZABLE`	Prevented	Prevented	Prevented	Audits / banking

Summary

All four transaction scenarios mapped to their concurrency strategy and isolation level.

Scenario	Concurrency Strategy	Isolation Level	Key Protection
1 — Placing an Order	`Pessimistic (SELECT FOR UPDATE)`	`READ COMMITTED`	Dirty reads, stock conflicts

Scenario	Concurrency Strategy	Isolation Level	Key Protection
2 — Completing a Delivery	`Optimistic (version timestamp)`	`READ COMMITTED`	Concurrent status updates
3 — Admin Adds Restaurant	`Pessimistic (intent lock)`	`READ COMMITTED`	Duplicate data
4 — Payment Refund	`Pessimistic (SELECT FOR UPDATE)`	`REPEATABLE READ`	Double refund prevention