B) Tech Tricks (450 Points) Time to step it up! These questions are more challenging and worth more points.

Q1 (70 points): Design a database schema for Bistro 92 to track users, orders, menu items, tables, and payments, optimized for fast queries.

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,          -- Auto-incrementing ID as the primary key
    full_name TEXT NOT NULL,        -- Full name of the user
    email TEXT UNIQUE NOT NULL,     -- Email, unique to prevent duplicates
    password TEXT NOT NULL,         -- Password (should be hashed in a real
application)
    is_verified BOOLEAN DEFAULT FALSE  -- Verification status, defaults to FALSE
);
```

```
-- Restaurant Table
CREATE TABLE Restaurant_Table (
    table_id SERIAL PRIMARY KEY,
    capacity INT NOT NULL,                -- Table's capacity (required)
    availability BOOLEAN NOT NULL DEFAULT TRUE  -- Table's availability status
(defaulted to TRUE)
);
```

```
-- Item Table
CREATE TABLE Item (
    item_id SERIAL PRIMARY KEY,                          -- Automatically
increments to provide unique item IDs
    item_name VARCHAR(255) NOT NULL,                      -- Item name
(required)
    description TEXT,                                     -- Description of the
item (optional)
    price DOUBLE PRECISION NOT NULL,                      -- Price of the item
(required, with support for decimals)
    rating DOUBLE PRECISION CHECK (rating >= 0 AND rating <= 5),  -- Rating with
validation between 0 and 5
    availability BOOLEAN NOT NULL DEFAULT TRUE,           -- Availability
status, defaulted to TRUE
    discount_percentage DOUBLE PRECISION CHECK (discount_percentage >= 0 AND
discount_percentage <= 100) -- Discount percentage (0 to 1000
```

```sql
-- OrderPackage Table (package_id is now a unique string)
CREATE TABLE OrderPackage (
    package_id VARCHAR(255) PRIMARY KEY,                    -- Unique package ID
(not serial)
    order_at TIMESTAMP NOT NULL,                          -- Date and time when the
order was placed
    completed_at TIMESTAMP,                               -- Date and time when the
order was completed
    total_price DOUBLE PRECISION NOT NULL,                -- Total price of the
package (order)
    status VARCHAR(50) CHECK (status IN ('pending', 'served')) NOT NULL, --
Package status (pending or served)
    table_id INT,                                         -- Foreign key to the
Restaurant_Table
    FOREIGN KEY (table_id) REFERENCES Restaurant_Table (table_id) ON DELETE SET
NULL  -- Table reference
);
```

```sql
-- Cart Table (referencing package_id as uniqueID is no longer necessary)
CREATE TABLE Cart (
    unique_serial SERIAL PRIMARY KEY,                     -- Unique serial number
for each cart item
    package_id VARCHAR(255) NOT NULL,                      -- Unique package ID from
OrderPackage table
    item_id INT NOT NULL,                                  -- Foreign key to Item
table (item_id)
    count INT NOT NULL,                                    -- Quantity of the item
in the cart
    FOREIGN KEY (package_id) REFERENCES OrderPackage (package_id) ON DELETE SET
NULL,   -- Package reference based on package_id
    FOREIGN KEY (item_id) REFERENCES Item (item_id) ON DELETE SET NULL          -
- Item reference
);
```

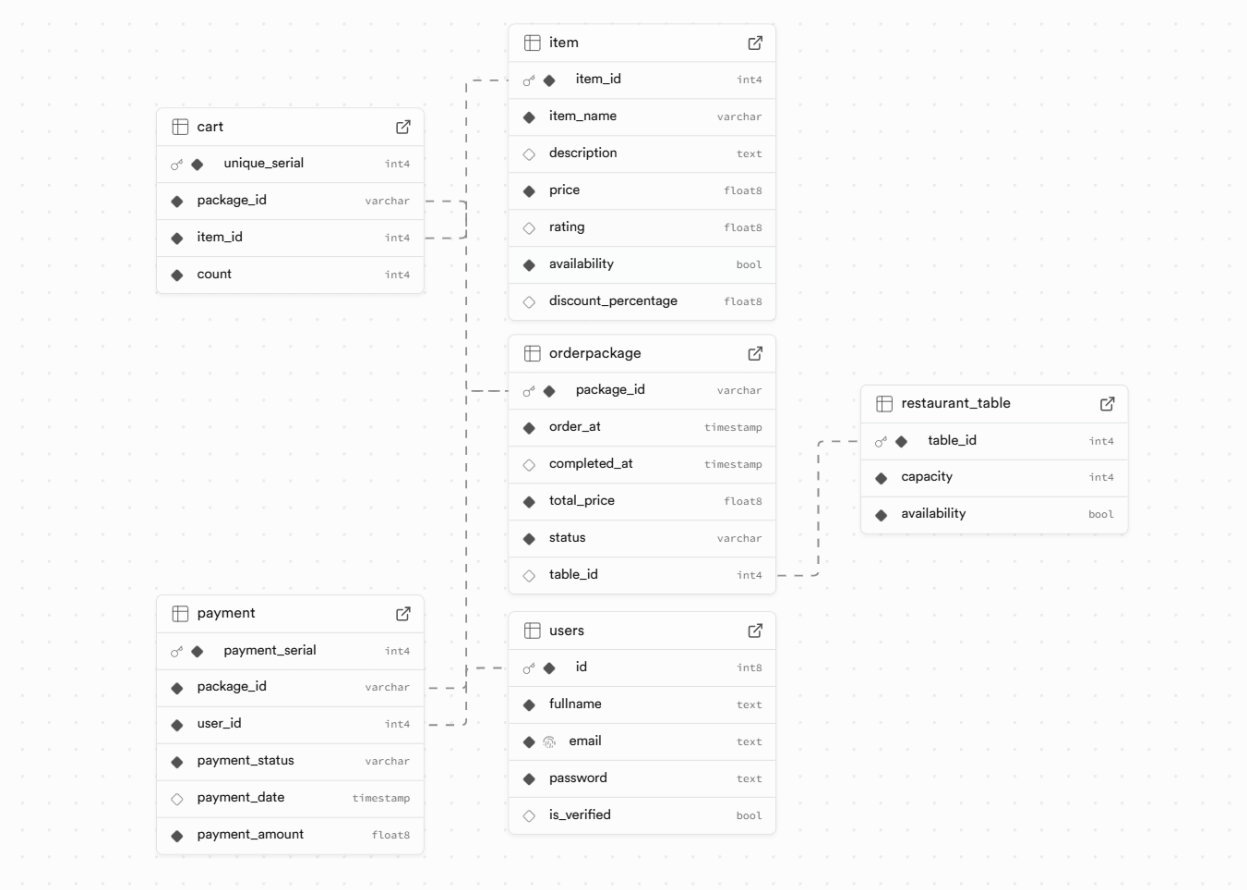
Made all many to many relation to 1-1 transforming relation into object

The schema is effectively in BCNF for practical purposes, though users.email introduces a minor theoretical deviation

Optimization by indexing:

```
-- Indexes for OrderPackage
CREATE INDEX idx_orderpackage_table_id ON OrderPackage (table_id);
CREATE INDEX idx_orderpackage_order_at ON OrderPackage (order_at);
CREATE INDEX idx_orderpackage_status ON OrderPackage (status);

-- Indexes for Cart
CREATE INDEX idx_cart_package_id ON Cart (package_id);
CREATE INDEX idx_cart_item_id ON Cart (item_id);
CREATE INDEX idx_cart_package_id_item_id ON Cart (package_id, item_id);
```



Q2 (80 points): Write an SQL query to retrieve all orders from the last hour, including table number, items ordered, and order time, optimized for speed.

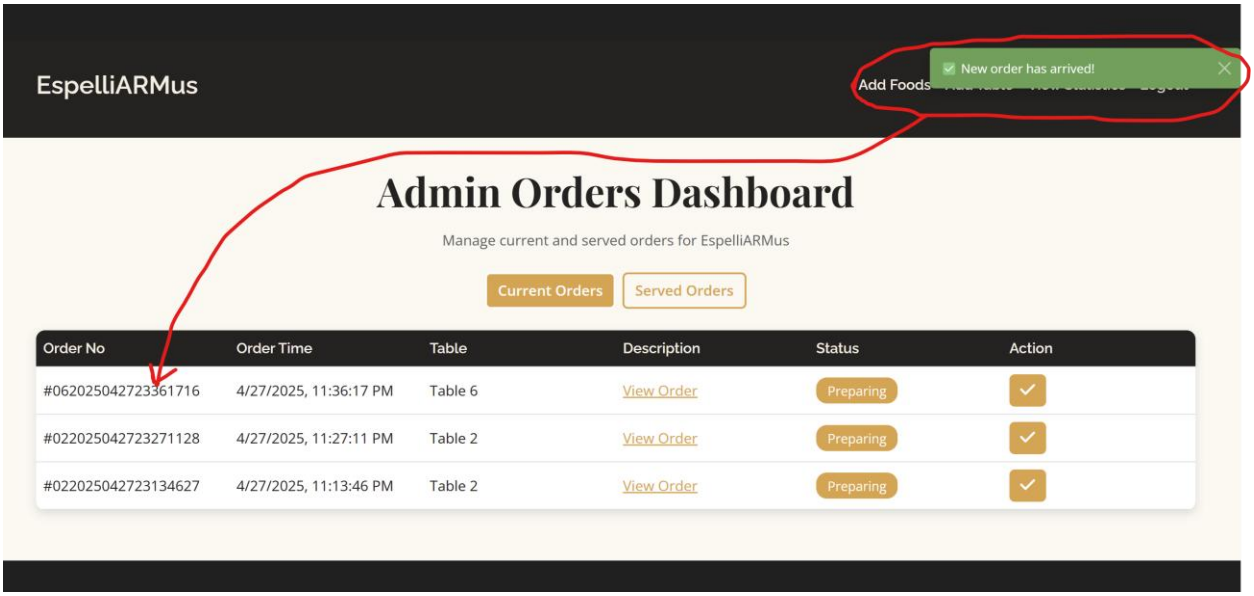
Optimization : Indexed,left joined,fetch Less number reduce load

Query:

```
SELECT
   op.package_id,
   op.order_at AS order_time,
   COALESCE(rt.table_id, 'N/A') AS table_number,
   STRING_AGG(CONCAT(i.item_name, ' (x', c.count, ')'), ', ') AS items_ordered
FROM OrderPackage op
LEFT JOIN Restaurant_Table rt ON op.table_id = rt.table_id
LEFT JOIN Cart c ON op.package_id = c.package_id
LEFT JOIN Item i ON c.item_id = i.item_id
WHERE op.order_at >= NOW() - INTERVAL '1 hour'
GROUP BY op.package_id, op.order_at, rt.table_id
ORDER BY op.order_at DESC;
```

Q3 (90 points): Implement a feature to notify kitchen staff in real-time when a new order is placed, and describe your tech stack.

Refer to Software demo:



Q4 (100 points): Describe a cloud-based system architecture for real-time updates, data storage, and smart pad communication, ensuring low latency and high availability.

This document outlines a cloud-based system architecture designed for real-time order updates, reliable data storage, and efficient communication between smart pad devices and the backend server, ensuring low latency and high availability.

1. Architecture Overview

The system uses a cloud-hosted Node.js server (Express.js) backend connected to a PostgreSQL database (Supabase) for data persistence. Smart Pads (ESP32/Arduino-based) communicate with the server over Wi-Fi using secure RESTful APIs. The admin dashboard is updated in real-time via AJAX polling every 5 seconds.

2. Components

- Node.js/Express Backend: Handles API requests from Smart Pads and Admin Dashboard.

- Supabase PostgreSQL: Cloud database for storing users, orders, tables, items, and payments.

- Smart Pads (Arduino/ESP32): Used by customers to place orders wirelessly.

- Admin Dashboard (Frontend): Displays current and served orders with real-time updates.

- Nodemailer Service: For sending OTPs during registration.

3. Real-Time Updates

- Smart Pads POST new orders to the backend.

- Backend updates the PostgreSQL database instantly.

- Admin dashboard polls every 5 seconds and displays a toast notification when new orders arrive.

- Optional: Future upgrade to WebSocket (Socket.io) for instant push notifications.

4. Data Storage

- Orders, Items, Tables, and Payments are stored in Supabase PostgreSQL tables.

- Each new order triggers an automatic backup into a local text file (items.txt) periodically.

- Server maintains a copy of important transaction records.

5. Smart Pad Communication

- Smart Pads connect via Wi-Fi to the internet.

- Use HTTP POST requests to send JSON payloads to endpoints like '/add-order-package' and '/add-cart-item'.

- Responses are acknowledged with success or error status codes.

- Lightweight payloads ensure minimal bandwidth usage.

6. Low Latency Measures

- Deployed in geographically close regions (e.g., Singapore AWS region).

- Lightweight API payloads and efficient database indexing.

- Backend optimized for concurrent lightweight HTTP requests.

7. High Availability Measures

- Cloud hosting with auto-scaling (Node.js on VPS or serverless deployment).

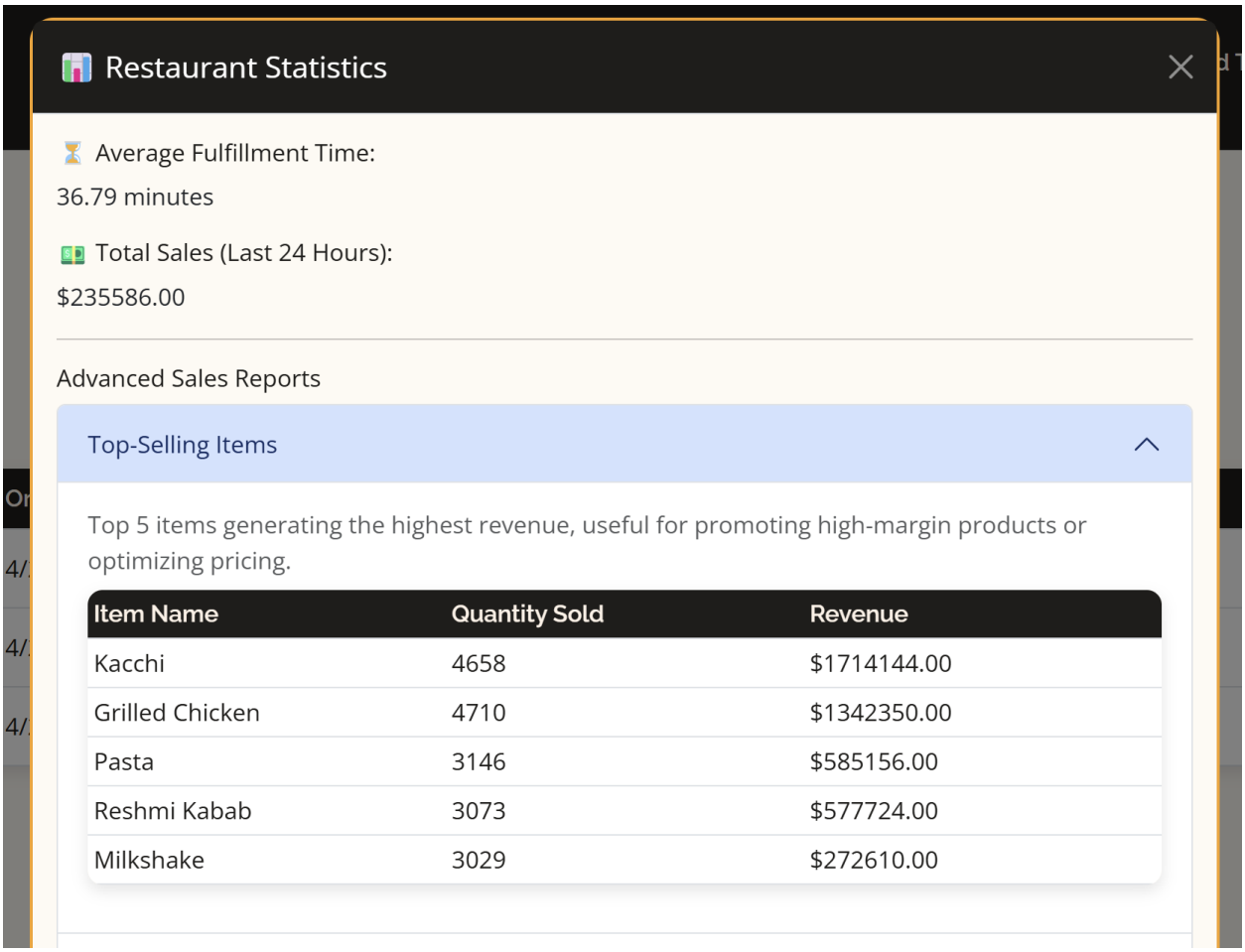
- Database replication and backups (provided by Supabase).

- Health monitoring services and automated restart scripts.

Q5 (110 points): Design a real-time dashboard showing pending orders, average fulfillment time, and total sales, specifying tools and justifying your choices.

We used 75% pern stack (postgres + express + (not reat,used raw html with css and js)+nodejs

Dashboard is given in third question;now this this sales reports:



Advanced Sales Reports

Top-Selling Items

Peak Order Times

Hours with the highest order volume, ideal for scheduling staff and preparing inventory.

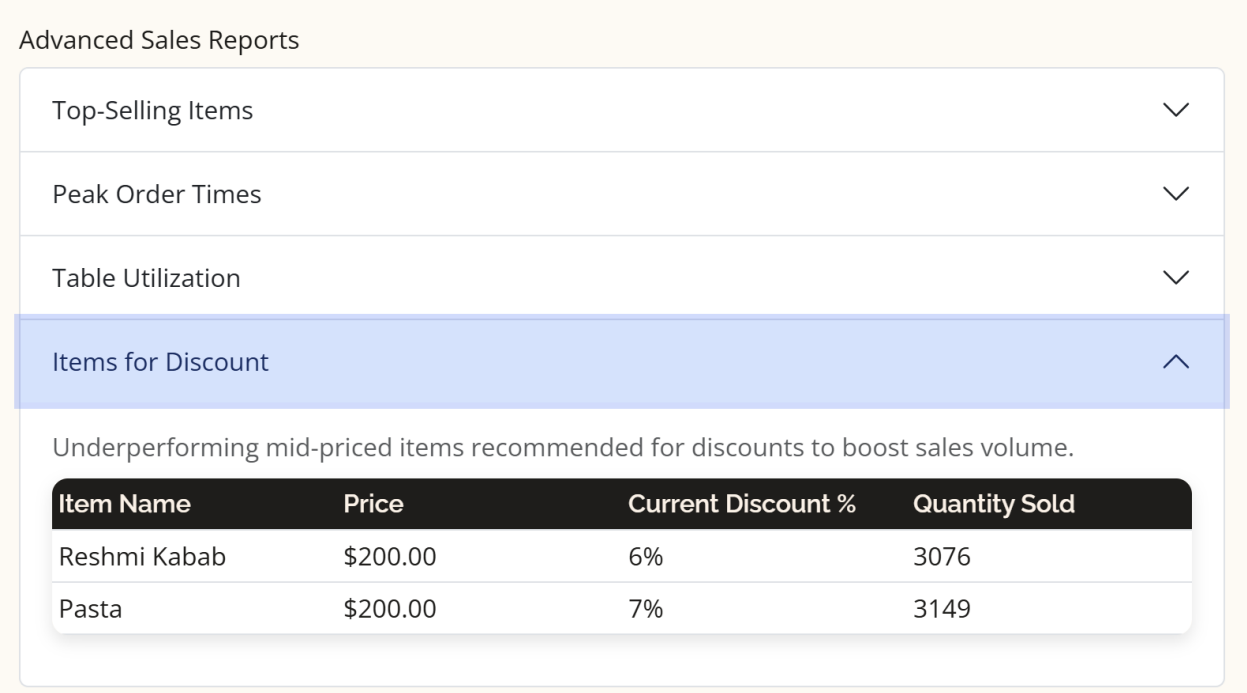
Hour	Orders	Revenue
11:00	872	$743973.00
12:00	854	$716548.00
14:00	813	$691467.00
13:00	785	$656128.00
21:00	573	$314286.00

Table Utilization

Items for Discount

Table usage by capacity, helping optimize seating arrangements and increase turnover.

Table ID	Capacity	Orders	Revenue
2	2	837	$556120.00
5	2	821	$533779.00
4	2	813	$527919.00
3	2	780	$505534.00
1	2	774	$513912.00
8	5	543	$402953.00
7	5	526	$384093.00
6	5	524	$389872.00
9	5	507	$374425.00
10	10	249	$303377.00



Advanced Queries for sale reports:

```
1. Top-Selling Items by Revenue

SELECT
    i.item_id,
    i.item_name,
    SUM(c.count) AS total_quantity_sold,
    SUM(c.count * i.price * (1 - i.discount_percentage / 100)) AS total_revenue
FROM Cart c
JOIN Item i ON c.item_id = i.item_id
JOIN OrderPackage op ON c.package_id = op.package_id
WHERE op.status = 'served'
GROUP BY i.item_id, i.item_name
ORDER BY total_revenue DESC
LIMIT 5;
```

```
2. Peak Order Times
SELECT
    EXTRACT(HOUR FROM order_at) AS order_hour,
    COUNT(DISTINCT package_id) AS total_orders,
    SUM(total_price) AS total_revenue
FROM OrderPackage
WHERE status = 'served'
GROUP BY order_hour
ORDER BY total_orders DESC;
```

```
3. Table Utilization Report
Purpose: Analyze which table sizes are most used to optimize seating arrangements
and increase turnover.

SELECT
    rt.table_id,
    rt.capacity,
    COUNT(op.package_id) AS total_orders,
    SUM(op.total_price) AS total_revenue
FROM Restaurant_Table rt
LEFT JOIN OrderPackage op ON rt.table_id = op.table_id AND op.status = 'served'
GROUP BY rt.table_id, rt.capacity
ORDER BY total_orders DESC;

4.Discount prediction

WITH ItemSales AS (
    SELECT
        i.item_id,
        i.item_name,
        i.price,
        i.discount_percentage,
        COALESCE(SUM(c.count), 0) AS total_quantity_sold,
        COALESCE(SUM(c.count * i.price * (1 - i.discount_percentage / 100)), 0)
AS total_revenue
    FROM Item i
    LEFT JOIN Cart c ON i.item_id = c.item_id
    LEFT JOIN OrderPackage op ON c.package_id = op.package_id AND op.status =
'served'
    GROUP BY i.item_id, i.item_name, i.price, i.discount_percentage
),
RankedItems AS (
```

```sql
    SELECT
        item_id,
        item_name,
        price,
        discount_percentage,
        total_quantity_sold,
        total_revenue,
        RANK() OVER (ORDER BY total_quantity_sold ASC) AS sales_rank
    FROM ItemSales
    WHERE total_quantity_sold > 0 -- Exclude items with no sales (may need
separate promotion)
)
SELECT
    item_id,
    item_name,
    price,
    discount_percentage,
    total_quantity_sold,
    total_revenue
FROM RankedItems
WHERE sales_rank <= 3 -- Focus on bottom 3 items by sales volume
AND discount_percentage < 15 -- Avoid items with already high discounts
AND price BETWEEN 150 AND 300 -- Target mid-priced items for price sensitivity
ORDER BY total_quantity_sold ASC;
```