FINAL PROJECT REPORT

Title: ***Agricultural Management System***

TEAM MEMBERS:

1) Vivek - 142301024

2) Havish - 142301003

3) Sai Varun - 142301040

Course: ***DS 3020***

❖ INTRODUCTION

Project Overview

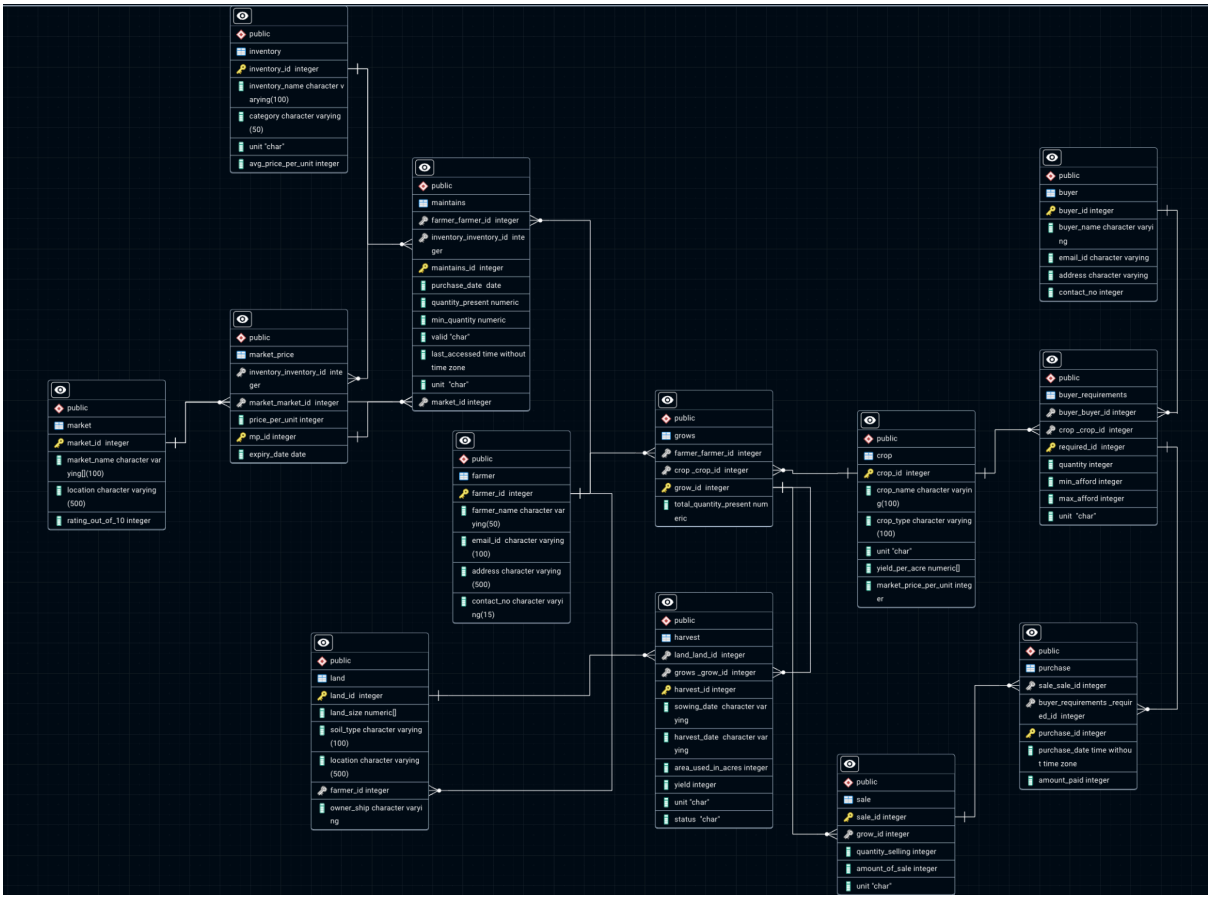
Agriculture is a vital sector, and farmers require an efficient way to manage crops, sales, and inventory. This project **digitizes farm management** by allowing farmers to track land ownership, monitor inventory, and sell crops efficiently.

Objectives

- Implement a **relational database** for farm management
- Develop SQL **queries and stored procedures**

➢ ER DIAGRAM & RELATIONAL MODEL:

Below is the **ER Diagram** of the Agricultural Management System:



Relational Model for the provided ER diagram:

Core Entities

- **Farmer** (farmer_id [PK], farmer_name, email_id, address, contact_no)
- **Land** (land_id [PK], land_size, soil_type, location, farmer_id [FK])
- **Crop** (crop_id [PK], crop_name, crop_type, unit, yield_per_acre, market_price_per_unit)

- **Inventory** (inventory_id [PK], inventory_name, category, unit, avg_price_per_unit)
- **Market** (market_id [PK], market_name, location, rating_out_of_10)
- **Buyer** (buyer_id [PK], buyer_name, email_id, address, contact_no)

Relationships

- **Maintains** (maintains_id [PK], farmer_farmer_id [FK], inventory_inventory_id [FK], purchase_date, quantity_present, min_quantity, valid, last_accessed, unit, market_id [FK])
- **Market_Price** (inventory_inventory_id [FK], market_market_id [FK], price_per_unit [FK], mq_id [FK], expiry_date)
- **Grows** (farmer_farmer_id [FK], crop_crop_id [FK], grow_id [PK], total_quantity_present)
- **Harvest** (harvest_id [PK], grows_grow_id [FK], land_land_id [FK], sowing_date, harvest_date, area_used_in_acres, yield, unit, status)
- **Buyer_Requirements** (buyer_buyer_id [PK], crop_crop_id [FK], required_id [FK], quantity, min_afford, max_afford, unit)
- **Purchase** (sale_sale_id [FK], buyer_requirements_required_id [FK], purchase_id [PK], purchase_date, amount_paid)
- **Sale** (sale_id [PK], grow_id [FK], quantity_selling, amount_of_sale, unit)

Entities and Attributes

Farmer

- Represents an individual who cultivates crops.

- **Attributes:**
  - farmer_id (PK): Unique identifier for each farmer.
  - farmer_name: Name of the farmer.
  - email_id: Email address of the farmer.
  - address: Residential or farm address (character varying 500).
  - contact_no: Phone number of the farmer (character varying 15).

Crop

- Represents different types of crops cultivated by farmers.
- **Attributes:**
  - crop_id (PK): Unique identifier for each crop.
  - crop_name: Name of the crop (character varying 100).
  - crop_type: Type of crop (character varying 100).
  - unit: Measurement unit for the crop (char).
  - yield_per_acre: Expected yield of the crop per acre of land (numeric).
  - market_price_per_unit: Current selling price per unit of the crop (integer).

Land

- Represents land plots used for cultivation.
- **Attributes:**
  - land_id (PK): Unique identifier for each land plot.
  - land_size: Size of the land (numeric).

    - soil_type: Type of soil on the land (character varying 100).
    - location: Geographical location of the land (character varying 500).
    - farmer_id (FK): Foreign key reference to the farmer who owns the land.
    - owner_ship: Type of ownership (character varying).

Inventory

- Represents farming-related stock.
- **Attributes:**
    - inventory_id (PK): Unique identifier for each inventory item.
    - inventory_name: Name of the inventory item (character varying 100).
    - category: Category of the inventory item (character varying 50).
    - unit: Measurement unit (char).
    - avg_price_per_unit: Average price per unit (integer).

Market

- Represents different markets where products are sold.
- **Attributes:**
    - market_id (PK): Unique identifier for each market.
    - market_name: Name of the market (character varying).
    - location: Location of the market (character varying 500).

    - rating_out_of_10: Rating of the market (integer).

Buyer

- Represents individuals or businesses purchasing crops from farmers.
- **Attributes:**
    - buyer_id (PK): Unique identifier for each buyer.
    - buyer_name: Name of the buyer (character varying).
    - email_id: Email address of the buyer (character varying).
    - address: Address of the buyer (character varying).
    - contact_no: Phone number of the buyer (integer).

Maintains

- Relationship between farmers and inventory.
- **Attributes:**
    - maintains_id (PK): Unique identifier for each maintenance record.
    - farmer_farmer_id (FK): Reference to farmer.
    - inventory_inventory_id (FK): Reference to inventory.
    - purchase_date: Date of purchase (date).
    - quantity_present: Current quantity available (numeric).
    - min_quantity: Minimum required quantity (numeric).
    - valid: Validity status (char).
    - last_accessed: Last access timestamp (time without time zone).

    - unit: Unit of measurement (char).
    - market_id (FK): Reference to market.

Market_Price

- Tracks prices of inventory items in different markets.
- **Attributes:**
    - inventory_inventory_id (FK): Reference to inventory.
    - market_market_id (FK): Reference to market.
    - price_per_unit: Price per unit (integer).
    - mq_id: Reference identifier (integer).
    - expiry_date: Expiration date (date).

Grows

- Captures which crops are grown by which farmers.
- **Attributes:**
    - farmer_farmer_id (FK): Reference to farmer.
    - crop_crop_id (FK): Reference to crop.
    - grow_id (PK): Unique identifier for each growing record.
    - total_quantity_present: Total quantity available (numeric).

Harvest

- Records details of crops harvested from specific lands.
- **Attributes:**
    - harvest_id (PK): Unique identifier for each harvest.
    - grows_grow_id (FK): Reference to grow record.

    - land_land_id (FK): Reference to land.
    - sowing_date: Date of sowing (character varying).
    - harvest_date: Date of harvest (character varying).
    - area_used_in_acres: Area used for cultivation (integer).
    - yield: Yield obtained (integer).
    - unit: Unit of measurement (char).
    - status: Status of the harvest (char).

Buyer_Requirements

- Indicates which crops buyers are interested in.
- **Attributes:**
    - buyer_buyer_id (FK): Reference to buyer.
    - crop_crop_id (FK): Reference to crop.
    - required_id (PK): Unique identifier for each requirement.
    - quantity: Required quantity (integer).
    - min_afford: Minimum affordable price (integer).
    - max_afford: Maximum affordable price (integer).
    - unit: Unit of measurement (char).

Purchase

- Records purchase transactions.
- **Attributes:**
    - sale_sale_id (FK): Reference to sale.
    - buyer_requirements_required_id (FK): Reference to buyer requirement.
    - purchase_id (PK): Unique identifier for each purchase.

- purchase_date: Date of purchase (time without time zone).
- amount_paid: Amount paid for the purchase (integer).

Sale

- Represents crop sales from farmers.
- **Attributes:**
  - sale_id (PK): Unique identifier for each sale transaction.
  - grow_id (FK): Reference to grow record.
  - quantity_selling: Quantity being sold (integer).
  - amount_of_sale: Total amount of the sale (integer).
  - unit: Unit of measurement (char).

➢**List of Tables:**

```
project=# \dt
               List of relations
 Schema |         Name          | Type  |  Owner
--------+-----------------------+-------+----------
 public | buyer                 | table | postgres
 public | buyer_requirements    | table | postgres
 public | crop                  | table | postgres
 public | farmer                | table | postgres
 public | grows                 | table | postgres
 public | harvest               | table | postgres
 public | inventory             | table | postgres
 public | land                  | table | postgres
 public | maintains             | table | postgres
 public | market                | table | postgres
 public | market_price          | table | postgres
 public | purchase              | table | postgres
 public | sale                  | table | postgres
(13 rows)
```

Tables with Constraints and Sample Data Population:

1.Farmer

```sql
CREATE TABLE farmer (
    farmer_id SERIAL PRIMARY KEY,
    farmer_name VARCHAR(50) NOT NULL,
    email_id VARCHAR(50),
    password VARCHAR(50),
    address VARCHAR(50),
    contact_no VARCHAR(15) NOT NULL
);
```

2. Crop

```sql
CREATE TABLE crop (
    crop_id SERIAL PRIMARY KEY,
    crop_name VARCHAR(20),
    crop_type VARCHAR(20),
    yield_per_acre NUMERIC(5,2),
    unit CHAR,
    market_price_per_unit INTEGER
);
```

3.Inventory

```sql
CREATE TABLE inventory (
    inventory_id SERIAL PRIMARY KEY,
    inventory_name VARCHAR(50) NOT NULL,
    category VARCHAR(50),
    avg_price_per_unit INTEGER,
    unit CHAR
);
```

4. Market

```sql
CREATE TABLE market (
    market_id SERIAL PRIMARY KEY,
    market_name VARCHAR(50) NOT NULL,
    location VARCHAR(50),
    rating_out_of_10 INTEGER
);
```

5.Market Price

```
CREATE TABLE market_price (
    mp_id SERIAL PRIMARY KEY,
    inventory_id INTEGER REFERENCES inventory(inventory_id),
    market_id INTEGER REFERENCES market(market_id),
    price_per_unit INTEGER,
    expiry_date DATE
);
```

6.Land

```
CREATE TABLE land (
    land_id SERIAL PRIMARY KEY,
    land_size NUMERIC(5,2),
    soil_type VARCHAR(50),
    location VARCHAR(50),
    farmer_id INTEGER REFERENCES farmer(farmer_id),
    ownership VARCHAR(50)
);
```

7.Buyer

```
CREATE TABLE buyer (
    buyer_id SERIAL PRIMARY KEY,
    buyer_name VARCHAR(50),
    email_id VARCHAR(50),
    address VARCHAR(50),
    contact_no INTEGER,
    password VARCHAR(50)
);
```

8.Buyer Requirements

```
CREATE TABLE buyer_requirements (
    required_id SERIAL PRIMARY KEY,
    buyer_id INTEGER REFERENCES buyer(buyer_id),
    crop_id INTEGER REFERENCES crop(crop_id),
    quantity NUMERIC(5,2),
    unit CHAR,
    min_afford INTEGER,
    max_afford INTEGER
);
```

9.Grows relation

```
CREATE TABLE grows (
    grow_id SERIAL PRIMARY KEY,
    farmer_id INTEGER REFERENCES farmer(farmer_id),
    crop_id INTEGER REFERENCES crop(crop_id),
    total_quantity_present NUMERIC
);
```

10.Harvest relation

```
CREATE TABLE harvest (
    harvest_id SERIAL PRIMARY KEY,
    land_id INTEGER REFERENCES land(land_id),
    grow_id INTEGER REFERENCES grows(grow_id),
    sowing_date DATE,
    harvest_date DATE,
    area_used_in_acres NUMERIC(5,2),
    yield NUMERIC(7,2),
    unit CHAR,
    status CHAR
);
```

11 Maintains relation

```
CREATE TABLE maintains (
    maintains_id SERIAL PRIMARY KEY,
    farmer_id INTEGER REFERENCES farmer(farmer_id),
    inventory_id INTEGER REFERENCES inventory(inventory_id),
    market_id INTEGER REFERENCES market(market_id),
    purchase_date DATE,
    quantity_present NUMERIC,
    min_quantity NUMERIC,
    unit CHAR,
    valid CHAR,
    last_accessed DATE
);
```

12.Sale

```
CREATE TABLE sale (
    sale_id SERIAL PRIMARY KEY,
    grow_id INTEGER REFERENCES grows(grow_id),
    quantity_selling INTEGER,
    unit CHAR,
    amount_of_sale INTEGER
);
```

13. Purchase

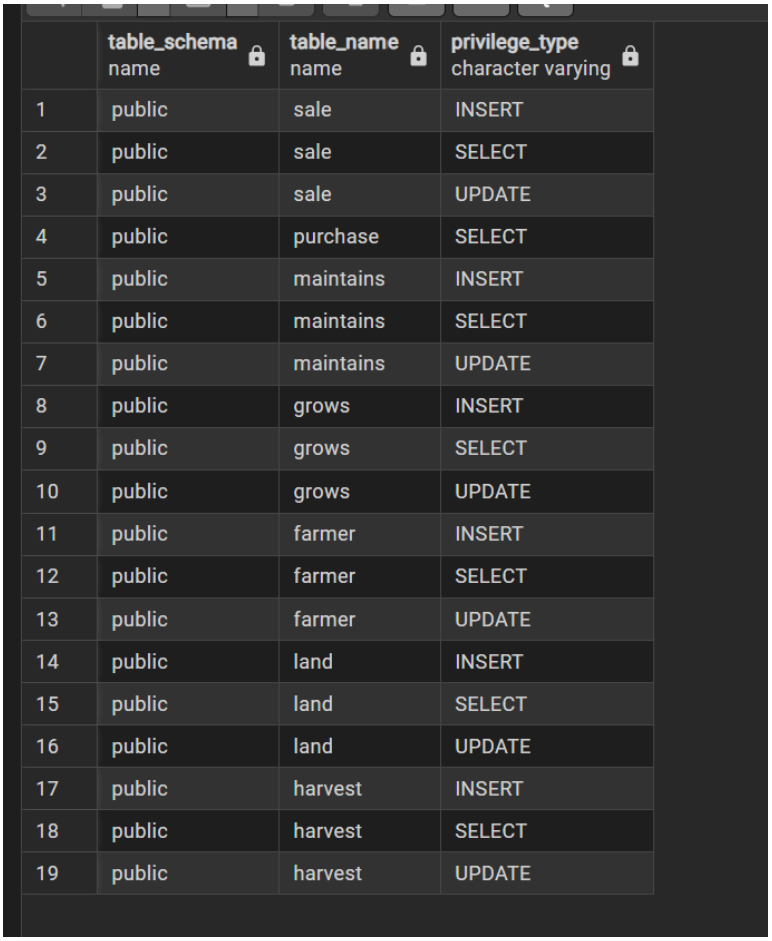
```
CREATE TABLE purchase (
    purchase_id SERIAL PRIMARY KEY,
    sale_id INTEGER REFERENCES sale(sale_id),
    required_id INTEGER REFERENCES buyer_requirements(required_id),
    purchase_date TIME WITHOUT TIME ZONE,
    amount_paid INTEGER
);
```

ROLES

1) **Farmer Role:**

Permissions: Read, insert, and update on farmer, land, grows, harvest, sale and maintains tables. Read only on purchase.

Purpose: Allows farmers to manage their personal information, land details, crops they grow, harvest records,check their transaction from purchase and inventory they maintain.



	table_schema name	table_name name	privilege_type character varying
1	public	sale	INSERT
2	public	sale	SELECT
3	public	sale	UPDATE
4	public	purchase	SELECT
5	public	maintains	INSERT
6	public	maintains	SELECT
7	public	maintains	UPDATE
8	public	grows	INSERT
9	public	grows	SELECT
10	public	grows	UPDATE
11	public	farmer	INSERT
12	public	farmer	SELECT
13	public	farmer	UPDATE
14	public	land	INSERT
15	public	land	SELECT
16	public	land	UPDATE
17	public	harvest	INSERT
18	public	harvest	SELECT
19	public	harvest	UPDATE

View under farmer_role:

Farmer_harvests
to display a farmer's crop and harvest data

```sql
CREATE VIEW farmer_harvests AS
SELECT
    h.harvest_id,
    g.farmer_id,
    c.crop_id,
    c.crop_name,
    h.land_id,
    h.area_used_in_acres,
    h.sowing_date,
    h.harvest_date,
    h.status,
    h.yield
FROM
    harvest h
JOIN
    grows g ON h.grow_id = g.grow_id
JOIN
    crop c ON g.crop_id = c.crop_id;

SELECT * FROM farmer_harvests WHERE farmer_id = 1 ORDER BY harvest_date DESC;
```

Output | Messages | Notifications

Showing rows: 1

harvest_id integer	farmer_id integer	crop_id integer	crop_name character varying (20)	land_id integer	area_used_in_acres numeric (5,2)	sowing_date date	harvest_date date	status character (1)	yield numeric (7,2)
70	1	2	Rice	26	3.45	2024-08-15	2025-04-14	P	[null]

Farmer_inventory

to show a farmer's inventory, including validity and market details.

```sql
CREATE VIEW farmer_inventory AS
SELECT
    m.maintains_id,
    m.farmer_id,
    m.inventory_id,
    i.inventory_name,
    m.market_id,
    mk.market_name,
    m.purchase_date,
    m.quantity_present,
    m.min_quantity,
    m.unit,
    m.valid,
    m.last_accessed,
    mp.expiry_date
FROM
    maintains m
JOIN
    inventory i ON m.inventory_id = i.inventory_id
JOIN
    market mk ON m.market_id = mk.market_id
LEFT JOIN
    market_price mp ON m.inventory_id = mp.inventory_id AND m.market_id = mp.market_id;

SELECT * FROM farmer_inventory WHERE farmer_id = 1 ORDER BY purchase_date DESC;
```

Output | Messages | Notifications

Showing rows: 1 to 5 | Page No: 1

maintains_id integer	farmer_id integer	inventory_id integer	inventory_name character varying (50)	market_id integer	market_name character varying (50)	purchase_date date	quantity_present numeric	min_quantity numeric	unit character (1)	valid character (1)	last_accessed date	expiry_date date
31	1	10	Irrigation Pipes	10	Village Supplies	2024-03-15	987.65	50.00	U	Y	2025-03-15	2026-01-15
12	1	4	Wheat Seeds A1	6	Farm Essential Store	2024-02-20	789.12	40.12	K	N	2025-03-10	2025-08-10
5	1	2	Pesticide X-40	1	Green Valley Market	2024-02-05	89.12	5.67	L	N	2025-01-20	2025-10-15
25	1	8	Maize Seeds M30	8	Country Market	2024-01-20	345.67	18.90	K	N	2025-03-05	2025-06-30
18	1	5	Rice Seeds R99	2	Farmers Central	2024-01-15	543.21	28.90	K	N	2025-03-25	2025-03-20

Farmer_sales

to display a farmer's active sales

```sql
CREATE VIEW farmer_sales AS
SELECT
    s.sale_id,
    g.farmer_id,
    s.grow_id,
    g.crop_id,
    c.crop_name,
    s.quantity_selling,
    s.unit,
    s.amount_of_sale
FROM
    sale s
JOIN
    grows g ON s.grow_id = g.grow_id
JOIN
    crop c ON g.crop_id = c.crop_id;

SELECT * FROM farmer_sales WHERE farmer_id = 1 ORDER BY sale_id DESC;
```

a Output | Messages | Notifications

SQL

sale_id integer	farmer_id integer	grow_id integer	crop_id integer	crop_name character varying (20)	quantity_selling integer	unit character (1)	amount_of_sale integer
2	1	2	4	Potato	200	K	1320
1	1	1	2	Rice	80	K	1440

2) **Buyer Role:**

Permissions: Read, insert, and update on buyer and buyer_requirements tables. Read only on sales and purchases.
Purpose: Enables buyers to manage their profile and specify their crop requirements and check their transaction history

	table_schema name	table_name name	privilege_type character varying
1	public	purchase	SELECT
2	public	sale	SELECT
3	public	buyer	INSERT
4	public	buyer	SELECT
5	public	buyer	UPDATE
6	public	buyer_requirements	INSERT
7	public	buyer_requirements	SELECT
8	public	buyer_requirements	UPDATE

Views under Buyer Role

Buyer_profile

To show buyer profile data

```
225 CREATE VIEW buyer_profile AS
226 SELECT
227     buyer_id,
228     buyer_name,
229     email_id,
230     contact_no,
231     address
232 FROM
233     buyer;
234
235 SELECT * FROM buyer_profile WHERE buyer_id = 1;
236
```

Data Output | Messages | Notifications

SQL

	buyer_id integer	buyer_name character varying (50)	email_id character varying (50)	contact_no integer	address character varying (50)
1	1	Fresh Foods	purchasing@freshfoods.com	5550201	100 Market Street, North City

Buyer_requirements

To show buyer crop requirements

```sql
CREATE VIEW buyer_requirement AS
SELECT
    br.required_id,
    br.buyer_id,
    b.buyer_name,
    br.crop_id,
    c.crop_name,
    br.quantity,
    br.unit,
    br.min_afford,
    br.max_afford
FROM
    buyer_requirements br
JOIN
    crop c ON br.crop_id = c.crop_id
JOIN
    buyer b ON br.buyer_id = b.buyer_id;


SELECT * FROM buyer_requirement WHERE buyer_id = 1 ORDER BY required_id DESC;
```

Output Messages Notifications

SQL

required_id integer	buyer_id integer	buyer_name character varying (50)	crop_id integer	crop_name character varying (20)	quantity numeric (5,2)	unit character (1)	min_afford integer	max_afford integer
23	1	Fresh Foods	8	Tomato	25.00	K	1400	1600
2	1	Fresh Foods	6	Soybean	50.00	K	1000	1300
1	1	Fresh Foods	4	Potato	50.00	K	1000	1300

3) Admin Role

Permissions: Read, insert, and update on market, market_price, crops and inventory tables; read-only on purchase, maintains, grows, farmer, land, harvest, buyer, and buyer_requirements tables.

Purpose: Allows admins to oversee market details, set prices, manage inventory, and monitor all other activities across the system.

```
project=# SELECT table_schema, table_name, privilege_type FROM information_schema.table_privileges WHERE grantee = 'admin_role';
 table_schema |     table_name     | privilege_type
--------------+--------------------+----------------
 public       | inventory          | INSERT
 public       | inventory          | SELECT
 public       | inventory          | UPDATE
 public       | maintains          | SELECT
 public       | grows              | SELECT
 public       | market             | INSERT
 public       | market             | SELECT
 public       | market             | UPDATE
 public       | market_price       | INSERT
 public       | market_price       | SELECT
 public       | market_price       | UPDATE
 public       | farmer             | SELECT
 public       | land               | SELECT
 public       | harvest            | SELECT
 public       | buyer              | SELECT
 public       | buyer_requirements | SELECT
(16 rows)
```

Views under Admin Role

To fetch all market_prices in market for **inventory**

```
ery  Query History

CREATE VIEW admin_market_prices AS
SELECT
    mp.inventory_id,
    i.inventory_name,
    mp.market_id,
    m.market_name,
    mp.price_per_unit,
    mp.expiry_date
FROM
    market_price mp
JOIN
    inventory i ON mp.inventory_id = i.inventory_id
JOIN
    market m ON mp.market_id = m.market_id;

select * from admin_market_prices
```

ta Output Messages Notifications

SQL Showing rows: 1 to

inventory_id integer	inventory_name character varying (50)	market_id integer	market_name character varying (50)	price_per_unit integer	expiry_date date
1	Fertilizer NPK	1	Green Valley Market	1250	2025-12-31
1	Fertilizer NPK	2	Farmers Central	1350	2025-12-01
1	Fertilizer NPK	12	Crop Market	1300	2025-12-10
1	Fertilizer NPK	4	AgriMart	1200	2025-12-15
2	Pesticide X-40	1	Green Valley Market	820	2025-10-15
2	Pesticide X-40	2	Farmers Central	700	2025-10-02

To see the details of inventory farmers has and suggest a good inventory for farmers

```
CREATE VIEW admin_maintains_monitor AS
SELECT
    m.maintains_id,
    m.farmer_id,
    f.farmer_name,
    m.inventory_id,
    i.inventory_name,
    m.market_id,
    mk.market_name,
    m.purchase_date,
    m.quantity_present,
    m.min_quantity,
    m.unit,
    m.valid,
    m.last_accessed
FROM
    maintains m
JOIN
    farmer f ON m.farmer_id = f.farmer_id
JOIN
    inventory i ON m.inventory_id = i.inventory_id
JOIN
    market mk ON m.market_id = mk.market_id;

select * from admin_maintains_monitor
```

Output Messages Notifications

SQL Showing rows: 1 to 33 Page No: 1 of 1

maintains_id integer	farmer_id integer	farmer_name character varying (50)	inventory_id integer	inventory_name character varying (50)	market_id integer	market_name character varying (50)	purchase_date date	quantity_present numeric	min_quantity numeric	unit character (1)	valid character (1)	last_accessed date
1	5	Robert Brown	1	Fertilizer NPK	1	Green Valley Market	2024-03-15	245.67	12.34	K	Y	2025-02-10
2	8	Linda Wilson	1	Fertilizer NPK	2	Farmers Central	2023-11-20	156.89	8.90	K	Y	2024-12-05
3	2	Maria Garcia	1	Fertilizer NPK	4	AgriMart	2024-01-10	378.45	25.67	K	Y	2025-03-15
4	11	Thomas Martinez	1	Fertilizer NPK	12	Crop Market	2023-06-25	512.34	15.43	K	Y	2024-11-30
6	7	James Miller	2	Pesticide X-40	7	Seeds & More	2023-09-15	423.78	33.21	L	Y	2025-10-15
7	3	David Johnson	2	Pesticide X-40	2	Farmers Central	2023-12-10	167.45	9.87	L	Y	2025-03-01
8	9	Michael Taylor	2	Pesticide X-40	9	Agricultural Hub	2024-03-01	298.56	20.14	L	N	2025-04-05
9	4	Sarah Williams	3	Tractor Fuel	4	AgriMart	2023-07-20	654.32	45.67	L	Y	2024-09-25

To fetch which farmer is growing what crops

```sql
CREATE VIEW admin_harvests_monitor AS
SELECT
    h.harvest_id,
    g.farmer_id,
    f.farmer_name,
    c.crop_id,
    c.crop_name,
    h.land_id,
    h.area_used_in_acres,
    h.sowing_date,
    h.harvest_date,
    h.status,
    h.yield
FROM
    harvest h
JOIN
    grows g ON h.grow_id = g.grow_id
JOIN
    crop c ON g.crop_id = c.crop_id
JOIN
    farmer f ON g.farmer_id = f.farmer_id;

select * from admin_harvests_monitor
```

Output Messages Notifications

Showing rows: 1 to 64 Page No: 1 of 1

harvest_id integer	farmer_id integer	farmer_name character varying (50)	crop_id integer	crop_name character varying (20)	land_id integer	area_used_in_acres numeric (5,2)	sowing_date date	harvest_date date	status character (1)	yield numeric (7,2)
6	1	John Smith	12	Barley	1	6.14	2023-06-15	2023-11-25	C	267.09
7	2	Maria Garcia	2	Rice	27	3.89	2024-04-05	2025-09-15	P	[null]
9	2	Maria Garcia	2	Rice	6	4.72	2023-09-05	2024-02-15	C	182.19
10	2	Maria Garcia	5	Cotton	27	1.95	2024-02-20	2025-07-30	P	[null]
11	2	Maria Garcia	5	Cotton	7	5.33	2023-07-15	2023-12-25	C	154.04
12	2	Maria Garcia	5	Cotton	6	3.67	2024-01-15	2025-06-25	P	[null]
13	3	David Johnson	3	Maize	8	2.41	2023-10-10	2025-03-20	C	126.04
14	3	David Johnson	11	Grapes	8	4.88	2024-03-15	2025-08-25	P	[null]

To see the buyer_requirements and give the sales accordingly

```sql
CREATE VIEW admin_requirements_monitor AS
SELECT
    br.required_id,
    br.buyer_id,
    b.buyer_name,
    br.crop_id,
    c.crop_name,
    br.quantity,
    br.unit,
    br.min_afford,
    br.max_afford
FROM
    buyer_requirements br
JOIN
    buyer b ON br.buyer_id = b.buyer_id
JOIN
    crop c ON br.crop_id = c.crop_id;

select * from admin_requirements_monitor
```

Output Messages Notifications

required_id integer	buyer_id integer	buyer_name character varying (50)	crop_id integer	crop_name character varying (20)	quantity numeric (5,2)	unit character varying (10)	min_afford integer	max_afford integer
2	1	Fresh Foods	6	Soybean	50.00	K	1000	1300
3	2	Grain Processors	1	Wheat	100.00	K	1700	1900
4	3	Farm to Market	8	Tomato	30.00	K	1400	1600
5	3	Farm to Market	11	Grapes	40.00	K	3600	3800
6	3	Farm to Market	5	Cotton	60.00	K	3200	3800
7	4	Wholesale Produce	9	Onion	45.00	K	1300	1500
8	4	Wholesale Produce	10	Apple	15.00	K	1300	1500
9	5	Food Distributors	3	Maize	80.00	K	1500	1700

Functions and their related triggers.

1. trigger_set_harvest_status_yield

Associated Function: set_harvest_status_yield()
Trigger Timing: BEFORE INSERT OR UPDATE on harvest table

Purpose: Automatically sets the status and yield fields in the harvest table based on the harvest_date and crop data.

Description:

- **What It Does**: This trigger ensures that when a new harvest record is inserted or an existing one is updated, the status is set to 'C' (Completed) if the harvest_date is on or before the current date, or 'P' (Pending) if it's in the future. It also calculates the yield by multiplying the area_used_in_acres by the yield_per_acre from the crop table (via grows) if the harvest is completed.
- **How It Works**:
    - **INSERT**:
        - Checks if harvest_date <= CURRENT_DATE.
        - If true (harvest is complete):
            - Retrieves yield_per_acre from the crop table using the crop_id linked through the grow_id.
            - If yield_per_acre and area_used_in_acres are not NULL, calculates yield = area_used_in_acres * yield_per_acre.
            - Sets status = 'C'.
        - If false (harvest is future):
            - Sets status = 'P'.
            - Leaves yield as is (could be NULL or user-provided).
    - **UPDATE**:
        - Stores the old yield for reference.
        - If harvest_date changes and status isn't already 'C':
            - Repeats the INSERT logic to update status and yield.
        - Ignores updates if status is already 'C' to prevent overwriting completed harvests.
- **Execution**: Runs before the row is inserted or updated in harvest, modifying NEW (the incoming row).
- **Role in System**: Ensures harvests are accurately marked as completed or pending and calculates yields automatically, reducing manual input errors. Supports farmers managing crops in /my_crops by providing consistent status and yield data.

```sql
-- 1)Create function for BEFORE INSERT/UPDATE to handle status and yield
CREATE OR REPLACE FUNCTION set_harvest_status_yield()
RETURNS TRIGGER AS $$
DECLARE
    ypa NUMERIC;
    old_yield NUMERIC;
BEGIN
    IF TG_OP = 'INSERT' THEN
        -- Check if harvest_date is on or before the current date
        IF NEW.harvest_date <= CURRENT_DATE THEN
            -- Get yield_per_acre from the crop table via grows
            SELECT yield_per_acre INTO ypa
            FROM crop
            WHERE crop_id = (SELECT crop_id FROM grows WHERE grow_id = NEW.grow_id);

            -- Calculate yield if yield_per_acre and area_used_in_acres are available
            IF ypa IS NOT NULL AND NEW.area_used_in_acres IS NOT NULL THEN
                NEW.yield := NEW.area_used_in_acres * ypa;
            END IF;

            -- Set status to 'C' (Completed)
            NEW.status := 'C';
        ELSE
            -- If harvest_date is in the future, set status to 'P' (Pending)
            NEW.status := 'P';
            -- Leave yield as is (could be NULL or a user-provided value)
        END IF;

        RETURN NEW;

    ELSIF TG_OP = 'UPDATE' THEN
        -- Store the old yield for adjustment
        old_yield := OLD.yield;

        -- Recalculate status and yield if harvest_date changes or status isn't already 'C'
        IF NEW.harvest_date != OLD.harvest_date AND NEW.status != 'C' THEN
            IF NEW.harvest_date <= CURRENT_DATE THEN
                -- Get yield_per_acre from the crop table via grows
                SELECT yield_per_acre INTO ypa
                FROM crop
                WHERE crop_id = (SELECT crop_id FROM grows WHERE grow_id = NEW.grow_id);

                -- Calculate yield if yield_per_acre and area_used_in_acres are available
                IF ypa IS NOT NULL AND NEW.area_used_in_acres IS NOT NULL THEN
                    NEW.yield := NEW.area_used_in_acres * ypa;
                END IF;

                -- Set status to 'C' (Completed)
                NEW.status := 'C';
            ELSE
                -- If harvest_date is in the future, set status to 'P' (Pending)
                NEW.status := 'P';
            END IF;
        END IF;

        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_set_harvest_status_yield
BEFORE INSERT OR UPDATE ON harvest
FOR EACH ROW
EXECUTE FUNCTION set_harvest_status_yield();
```

2. trigger_update_grows_total_quantity

Associated Function: update_grows_total_quantity()
Trigger Timing: AFTER INSERT OR UPDATE on harvest table

Purpose: Updates the total_quantity_present in the grows table to reflect the yield from new or updated harvests.

Description:

- **What It Does**: This trigger updates the total_quantity_present in the grows table by adding the yield from a new harvest or adjusting it when a harvest's yield changes, ensuring the farmer's crop quantity is accurate.
- **How It Works**:
  - **INSERT**:
    - If the new harvest's yield is not NULL, adds yield to total_quantity_present in the grows table for the corresponding grow_id.
  - **UPDATE**:
    - Stores the old yield.
    - If yield changes (OLD.yield differs from NEW.yield):
      - Subtracts the old yield (or 0 if NULL) and adds the new yield (or 0 if NULL) to total_quantity_present in grows.
  - **Execution**: Runs after the row is inserted or updated in harvest, ensuring the harvest data is saved before updating grows.
- **Role in System**: Keeps the grows table's total_quantity_present in sync with harvest yields, critical for tracking available crop quantities for sales (e.g., /my_sales). Supports farmers by ensuring accurate inventory for transactions.

```sql
-- 2) Create function for AFTER INSERT/UPDATE to update total_quantity_present
CREATE OR REPLACE FUNCTION update_grows_total_quantity()
RETURNS TRIGGER AS $$
DECLARE
    old_yield NUMERIC;
BEGIN
    IF TG_OP = 'INSERT' THEN
        -- Update total_quantity_present in grows if yield is calculated or provided
        IF NEW.yield IS NOT NULL THEN
            UPDATE grows
            SET total_quantity_present = total_quantity_present + NEW.yield
            WHERE grow_id = NEW.grow_id;
        END IF;

        RETURN NEW;

    ELSIF TG_OP = 'UPDATE' THEN
        -- Store the old yield for adjustment
        old_yield := OLD.yield;

        -- Update total_quantity_present in grows if yield changes
        IF (OLD.yield IS DISTINCT FROM NEW.yield) THEN
            UPDATE grows
            SET total_quantity_present = total_quantity_present - COALESCE(old_yield, 0) + COALESCE(NEW.yield, 0)
            WHERE grow_id = NEW.grow_id;
        END IF;

        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER trigger_update_grows_total_quantity
AFTER INSERT OR UPDATE ON harvest
FOR EACH ROW
EXECUTE FUNCTION update_grows_total_quantity();
```

3. trg_check_area_used

Associated Function: check_area_used_limit()
Trigger Timing: BEFORE INSERT on harvest table

Purpose: Ensures the area_used_in_acres for a new harvest does not exceed the remaining available land area.

Description:

- **What It Does**: This trigger validates that the land area used for a new harvest (area_used_in_acres) does not exceed the remaining land area for the specified land_id, preventing over-allocation of land.
- **How It Works**:
    - Retrieves the land_size from the land table for the given land_id.
    - Raises an exception if:
        - The land_id doesn't exist.
        - land_size is NULL.
        - area_used_in_acres is NULL or negative.
    - Calculates the total area_used_in_acres already allocated for the land_id from existing harvest records.
    - Computes remaining_land = land_size - total_area_used.
    - If NEW.area_used_in_acres > remaining_land, raises an exception to block the insert.
    - If valid, allows the insert to proceed.
- **Execution**: Runs before inserting a row in harvest, checking NEW (the incoming row).
- **Role in System**: Enforces land usage constraints, ensuring farmers don't allocate more land than available when adding harvests in /my_crops. Critical for data integrity in land management.

```sql
CREATE OR REPLACE FUNCTION check_area_used_limit()
RETURNS TRIGGER AS $$
DECLARE
    land_sz NUMERIC;
    total_area_used NUMERIC;
    remaining_land NUMERIC;
BEGIN
    SELECT land_size INTO land_sz
    FROM land
    WHERE land_id = NEW.land_id;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'Land with land_id % does not exist.', NEW.land_id;
    END IF;

    IF land_sz IS NULL THEN
        RAISE EXCEPTION 'Land size is NULL for land_id %.', NEW.land_id;
    END IF;

    IF NEW.area_used_in_acres IS NULL THEN
        RAISE EXCEPTION 'Area used in acres cannot be NULL for land_id %.', NEW.land_id;
    END IF;

    IF NEW.area_used_in_acres < 0 THEN
        RAISE EXCEPTION 'Area used in acres cannot be negative for land_id %.', NEW.land_id;
    END IF;

    SELECT COALESCE(SUM(area_used_in_acres), 0) INTO total_area_used
    FROM harvest
    WHERE land_id = NEW.land_id;

    remaining_land := land_sz - total_area_used;

    IF NEW.area_used_in_acres > remaining_land THEN
        RAISE EXCEPTION 'Used area (% acres) exceeds remaining land (% acres) for land_id %.',
            NEW.area_used_in_acres, remaining_land, NEW.land_id;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_check_area_used
BEFORE INSERT ON harvest
FOR EACH ROW
EXECUTE FUNCTION check_area_used_limit();
```

4. trg_check_validity

Associated Function: update_validity_in_maintains()
Trigger Timing: BEFORE INSERT OR UPDATE on maintains table

Purpose: Sets the valid field in maintains based on the inventory's expiry_date compared to CURRENT_DATE.

Description:

- **What It Does**: This trigger checks the validity of inventory in the maintains table by comparing CURRENT_DATE to the expiry_date from market_price, setting valid to 'Y' (valid) or 'N' (invalid).
- **How It Works**:
  - Retrieves the expiry_date from market_price using NEW.inventory_id and NEW.market_id.
  - If CURRENT_DATE >= COALESCE(expiry_date, '9999-12-31'):
    - Sets NEW.valid = 'N' (invalid).
  - Otherwise:
    - Sets NEW.valid = 'Y' (valid).
  - The COALESCE ensures that if expiry_date is NULL, the inventory is considered valid (far future date).
- **Execution**: Runs before inserting or updating a row in maintains, modifying NEW.
- **Role in System**: Automates inventory validity checks for farmers in /my_inventory and admins in /admin_view_maintains. Ensures only valid inventory is used in sales or other operations, addressing your previous issue with mp_id errors (fixed by using inventory_id and market_id).

```
-- 4) this trigger is used to check the validity for an inventory .
CREATE OR REPLACE FUNCTION update_validity_in_maintains()
RETURNS TRIGGER AS $$
DECLARE
    exp_date DATE;
BEGIN
    -- Fetch expiry_date from market_price using inventory_id and market_id
    SELECT expiry_date INTO exp_date
    FROM market_price
    WHERE inventory_id = NEW.inventory_id AND market_id = NEW.market_id;

    -- Set valid based on CURRENT_DATE
    IF CURRENT_DATE >= COALESCE(exp_date, '9999-12-31') THEN
        NEW.valid := 'N';
    ELSE
        NEW.valid := 'Y';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Create the trigger
CREATE TRIGGER trg_check_validity
BEFORE INSERT OR UPDATE ON maintains
FOR EACH ROW
EXECUTE FUNCTION update_validity_in_maintains();
```

5. trg_restore_quantity_on_sale_delete

Associated Function: restore_quantity_after_sale_delete()
Trigger Timing: AFTER DELETE on sale table

Purpose: Restores the total_quantity_present in grows when a sale is deleted, but only if the sale hasn't been purchased.

Description:

- **What It Does**: This trigger ensures that if a sale is deleted and no purchases are associated with it, the quantity_selling is added back to total_quantity_present in the grows table, maintaining accurate crop quantities.
- **How It Works**:
  - Checks if the deleted sale (OLD.sale_id) exists in the purchase table by counting matching records.
  - If no purchases (is_purchased = 0):
    - Updates grows by adding OLD.quantity_selling to total_quantity_present for the corresponding grow_id.
  - If purchased, no action is taken (quantity remains deducted).
- **Execution**: Runs after a row is deleted from sale, using OLD (the deleted row).
- **Role in System**: Supports the /delete_sale route by restoring crop quantities when unsold sales are removed, ensuring farmers can re-list crops. Prevents quantity restoration for sold crops, maintaining data consistency.

```
183  -- 5) This is a function which restores the total_quantity_present if the sale is deleted.
184  CREATE OR REPLACE FUNCTION restore_quantity_after_sale_delete()
185  RETURNS TRIGGER AS $$
186  DECLARE
187      is_purchased INT;
188  BEGIN
189      -- Check if the sale exists in the purchase table
190      SELECT COUNT(*) INTO is_purchased FROM purchase WHERE sale_id = OLD.sale_id;
191
192      -- If not purchased, restore quantity
193      IF is_purchased = 0 THEN
194          UPDATE grows
195          SET total_quantity_present = total_quantity_present + OLD.quantity_selling
196          WHERE grow_id = OLD.grow_id;
197      END IF;
198
199      RETURN OLD;
200  END;
201  $$ LANGUAGE plpgsql;
202
203  CREATE TRIGGER trg_restore_quantity_on_sale_delete
204  AFTER DELETE ON sale
205  FOR EACH ROW
206  EXECUTE FUNCTION restore_quantity_after_sale_delete();
```

6. trg_reduce_quantity_on_sale_insert

Associated Function: reduce_quantity_after_sale_insert()
Trigger Timing: AFTER INSERT on sale table

Purpose: Reduces the total_quantity_present in grows when a new sale is added.

Description:

- **What It Does**: This trigger deducts the quantity_selling from total_quantity_present in the grows table when a farmer adds a sale, ensuring the available crop quantity is updated.
- **How It Works**:
    - Updates grows by subtracting NEW.quantity_selling from total_quantity_present for the corresponding grow_id.
- **Execution**: Runs after inserting a row in sale, using NEW.
- **Role in System**: Supports the /add_sale route (or similar) by reducing available crop quantities when sales are listed, ensuring farmers don't oversell. Works with trg_restore_quantity_on_sale_delete to maintain quantity consistency.

```
208  -- 6) Create a new function to reduce quantity after a sale is added
209  CREATE OR REPLACE FUNCTION reduce_quantity_after_sale_insert()
210  RETURNS TRIGGER AS $$
211  BEGIN
212      -- Reduce the total_quantity_present by the quantity_selling
213      UPDATE grows
214      SET total_quantity_present = total_quantity_present - NEW.quantity_selling
215      WHERE grow_id = NEW.grow_id;
216
217      RETURN NEW;
218  END;
219  $$ LANGUAGE plpgsql;
220
221  -- Create the trigger for sale insertion
222  CREATE TRIGGER trg_reduce_quantity_on_sale_insert
223  AFTER INSERT ON sale
224  FOR EACH ROW
225  EXECUTE FUNCTION reduce_quantity_after_sale_insert();
```

7. sale_update_trigger

Associated Function: update_sale_after_purchase()
Trigger Timing: AFTER INSERT on purchase table

Purpose: Updates or deletes a sale after a purchase is made, adjusting quantity_selling and amount_of_sale based on the purchased quantity.

Description:

- **What It Does**: This trigger ensures that when a buyer makes a purchase, the corresponding sale's quantity_selling and amount_of_sale are updated to reflect the remaining quantity, or the sale is deleted if fully purchased.
- **How It Works**:
    - Retrieves the quantity from buyer_requirements using NEW.required_id (the purchase's requirement ID).
    - Raises an exception if quantity is NULL.
    - Retrieves the current quantity_selling and amount_of_sale from sale using NEW.sale_id.
    - Raises an exception if quantity_selling or amount_of_sale is NULL or if quantity_selling < bought_quantity (insufficient stock).
    - If quantity_selling = bought_quantity:
        - Deletes the sale (fully purchased).
    - Otherwise:
        - Updates the sale:
            - quantity_selling = old_quantity_selling - bought_quantity.
            - amount_of_sale = old_amount_of_sale * (remaining_quantity / old_quantity_selling) (proportionally scales the amount).
- **Execution**: Runs after inserting a row in purchase, using NEW.
- **Role in System**: Supports the /buyer_transaction or purchase routes by updating sales after purchases, ensuring accurate sale data. Addresses your previous sale.amount_of_sale NULL issue by validating data and maintaining proportional amounts.

```sql
-- 7) Updating sale quantity and amount after the sale .
CREATE OR REPLACE FUNCTION update_sale_after_purchase()
RETURNS TRIGGER AS $$
DECLARE
    bought_quantity NUMERIC;
    old_quantity_selling NUMERIC;
    old_amount_of_sale NUMERIC;
BEGIN
    SELECT quantity INTO bought_quantity
    FROM buyer_requirements
    WHERE required_id = NEW.required_id;

    IF bought_quantity IS NULL THEN
        RAISE EXCEPTION 'Quantity not found for required_id %', NEW.required_id;
    END IF;

    SELECT quantity_selling, amount_of_sale INTO old_quantity_selling, old_amount_of_sale
    FROM sale
    WHERE sale_id = NEW.sale_id;

    IF old_quantity_selling IS NULL OR old_amount_of_sale IS NULL THEN
        RAISE EXCEPTION 'Invalid sale data for sale_id %: quantity_selling or amount_of_sale is NULL', NEW.sale_id;
    END IF;

    IF old_quantity_selling < bought_quantity THEN
        RAISE EXCEPTION 'Insufficient quantity available for sale_id %', NEW.sale_id;
    END IF;

    IF old_quantity_selling = bought_quantity THEN
        DELETE FROM sale WHERE sale_id = NEW.sale_id;
    ELSE
        UPDATE sale
        SET
            quantity_selling = old_quantity_selling - bought_quantity,
            amount_of_sale = old_amount_of_sale * ((old_quantity_selling - bought_quantity)::NUMERIC / old_quantity_selling)
        WHERE sale_id = NEW.sale_id;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER sale_update_trigger
AFTER INSERT ON purchase
FOR EACH ROW
EXECUTE FUNCTION update_sale_after_purchase();
```

Some more Additional Functions with Output:

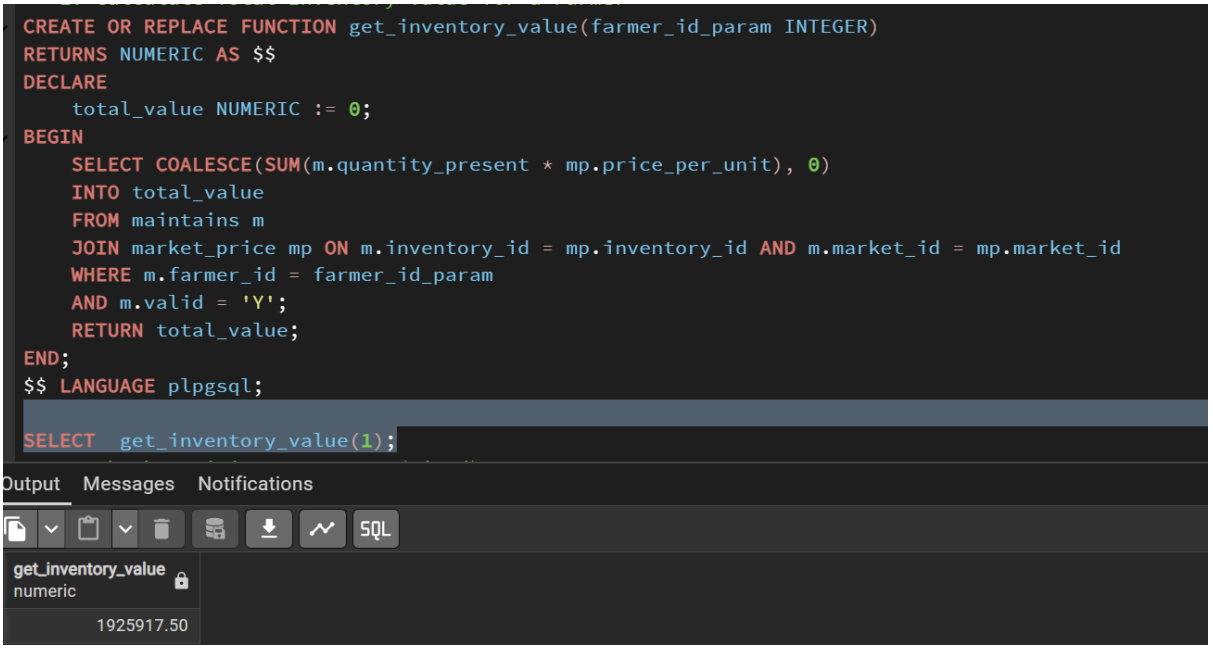
1. Get Inventory Value

Functionality: Calculates the total monetary value of a farmer's inventory by multiplying the quantity_present in the maintains table by the corresponding price_per_unit from the market_price table. It only considers valid inventory items (valid = 'Y').

Input: farmer_id_param - The unique identifier of the farmer.

Output: A single NUMERIC value representing the total inventory value.

Use Case: Used in the my_inventory route to display the total value of a farmer's inventory on their dashboard, helping them assess their asset worth.



2. Check Expiring Inventory

Functionality: Retrieves a list of inventory items maintained by a farmer, including their names, associated market names, expiry dates, and a status ('Expired', 'Expiring Soon', or 'Valid') based on the expiry_date from the market_price table. It filters for valid items (valid = 'Y') and uses a CASE statement to determine the status.

Input: farmer_id_param - The unique identifier of the farmer.

Output: A table with columns: maintains_id (INTEGER), inventory_name (VARCHAR), market_name (VARCHAR), expiry_date (DATE), status (VARCHAR).

Use Case: Provides alerts on the my_inventory page to notify farmers of expiring or expired inventory, enabling timely restocking or disposal decisions.

```sql
-- 2. Check Expiring Inventory (Fixed)
CREATE OR REPLACE FUNCTION check_inventory_expiry(farmer_id_param INTEGER)
RETURNS TABLE (
    maintains_id INTEGER,
    inventory_name VARCHAR,
    market_name VARCHAR,
    expiry_date DATE,
    status VARCHAR
) AS $$
BEGIN
    RETURN QUERY
    SELECT
        m.maintains_id,
        i.inventory_name,
        mk.market_name,
        mp.expiry_date,
        CASE
            WHEN mp.expiry_date < CURRENT_DATE THEN 'Expired'::VARCHAR
            WHEN mp.expiry_date <= CURRENT_DATE + INTERVAL '30 days' THEN 'Expiring Soon'::VARCHAR
            ELSE 'Valid'::VARCHAR
        END AS status
    FROM maintains m
    JOIN inventory i ON m.inventory_id = i.inventory_id
    JOIN market mk ON m.market_id = mk.market_id
    JOIN market_price mp ON m.inventory_id = mp.inventory_id AND m.market_id = mp.market_id
    WHERE m.farmer_id = farmer_id_param
    AND m.valid = 'Y';
END;
$$ LANGUAGE plpgsql;

SELECT * FROM check_inventory_expiry(3);
```

Output Messages Notifications

SQL

maintains_id integer	inventory_name character varying	market_name character varying	expiry_date date	status character varying
7	Pesticide X-40	Farmers Central	2025-10-02	Valid
20	Organic Compost	Green Valley Market	2025-11-16	Valid
33	Harvester Parts	Farm Equipment Center	2026-03-24	Valid

3. Get Farmer's Active Harvests

Functionality: Retrieves details of all active (pending) harvests for a farmer, identified by status = 'P'. It joins the harvest, grows, and crop tables to provide harvest ID, crop name, land ID, sowing date, harvest date, and area used.

Input: farmer_id_param - The unique identifier of the farmer.

Output: A table with columns: harvest_id (INTEGER), crop_name (VARCHAR), land_id (INTEGER), sowing_date (DATE), harvest_date (DATE), area_used_in_acres (NUMERIC).

Use Case: Populates the my_crops page with ongoing harvest information, allowing farmers to monitor their current growing activities.

```sql
CREATE OR REPLACE FUNCTION get_active_harvests(farmer_id_param INTEGER)
RETURNS TABLE (
    harvest_id INTEGER,
    crop_name VARCHAR,
    land_id INTEGER,
    sowing_date DATE,
    harvest_date DATE,
    area_used_in_acres NUMERIC
) AS $$
BEGIN
    RETURN QUERY
    SELECT
        h.harvest_id,
        c.crop_name,
        h.land_id,
        h.sowing_date,
        h.harvest_date,
        h.area_used_in_acres
    FROM harvest h
    JOIN grows g ON h.grow_id = g.grow_id
    JOIN crop c ON g.crop_id = c.crop_id
    WHERE g.farmer_id = farmer_id_param
    AND h.status = 'P';
END;
$$ LANGUAGE plpgsql;

SELECT * FROM get_active_harvests(1);
```

Output Messages Notifications

SQL

harvest_id integer	crop_name character varying	land_id integer	sowing_date date	harvest_date date	area_used_in_acres numeric
70	Rice	26	2024-08-15	2025-04-14	3.45

4. Calculate Total Sales Revenue for a Farmer

Functionality:.Calculates the total revenue from sales for a farmer by summing the amount_of_sale from the sale table, linked via the grows table.

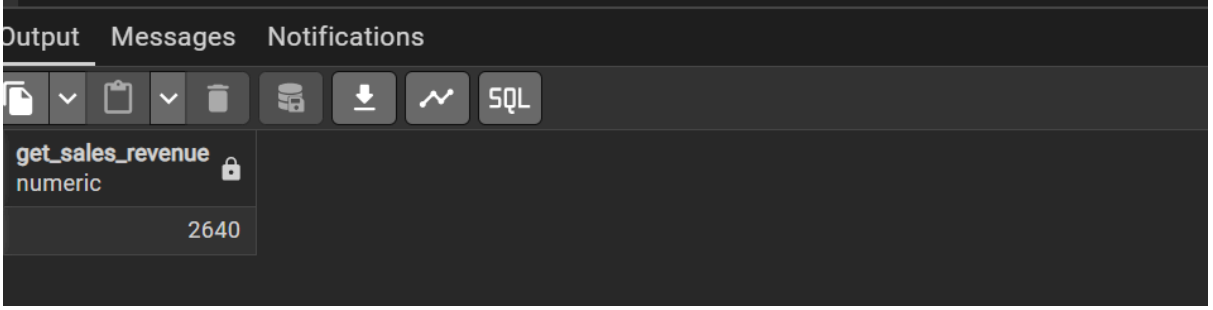
Input: farmer_id_param - The unique identifier of the farmer.

Output: A single NUMERIC value representing the total sales revenue.

Use Case: Displays revenue statistics on the my_sales or transaction_history page, helping farmers track their earnings.

```sql
CREATE OR REPLACE FUNCTION get_sales_revenue(farmer_id_param INTEGER)
RETURNS NUMERIC AS $$
DECLARE
    total_revenue NUMERIC := 0;
BEGIN
    SELECT COALESCE(SUM(s.amount_of_sale), 0)
    INTO total_revenue
    FROM sale s
    JOIN grows g ON s.grow_id = g.grow_id
    WHERE g.farmer_id = farmer_id_param;
    RETURN total_revenue;
END;
$$ LANGUAGE plpgsql;

SELECT * FROM get_sales_revenue(2);
```



5. Validate Farmer Land Ownership

Functionality: Checks if a specific land record belongs to the given farmer by verifying the farmer_id in the land table against the provided land_id.

Input: land_id_param - The unique identifier of the land; farmer_id_param - The unique identifier of the farmer.

Output: A BOOLEAN value (TRUE if the farmer owns the land, FALSE otherwise).

Use Case: Used in add_crop, update_land, and delete_land routes to prevent unauthorized modifications to land records.

```sql
-- 7. Validate Farmer Land Ownership
CREATE OR REPLACE FUNCTION validate_land_ownership(land_id_param INTEGER, farmer_id_param INTEGER)
RETURNS BOOLEAN AS $$
DECLARE
    exists BOOLEAN;
BEGIN
    SELECT EXISTS (
        SELECT 1
        FROM land
        WHERE land_id = land_id_param
        AND farmer_id = farmer_id_param
    ) INTO exists;
    RETURN exists;
END;
$$ LANGUAGE plpgsql;

SELECT * FROM validate_land_ownership(2, 5);
```

Output Messages Notifications

SQL

validate_land_ownership
boolean

false

Another example(true case):

```sql
SELECT * FROM validate_land_ownership(1, 1);
```

Output Messages Notifications

SQL

validate_land_ownership
boolean

true

6. Suggest Crops for a Buyer

Functionality: Recommends crops to a buyer based on their past requirements, joining the buyer_requirements, crop, grows, and sale tables. It calculates the total quantity_selling for each crop, filters for available quantities (> 0), and returns the top 5 crops by availability.

Input: buyer_id_param - The unique identifier of the buyer.

Output: A table with columns: crop_id (INTEGER), crop_name (VARCHAR), available_quantity (INTEGER).

Use Case: Enhances the purchase page with personalized crop suggestions, improving buyer decision-making.

```sql
-- 10. Suggest Crops for a Buyer
CREATE OR REPLACE FUNCTION suggest_crops_for_buyer(buyer_id_param INTEGER)
RETURNS TABLE (
    crop_id INTEGER,
    crop_name VARCHAR,
    available_quantity INTEGER
) AS $$
BEGIN
    RETURN QUERY
    SELECT DISTINCT
        c.crop_id,
        c.crop_name,
        COALESCE(SUM(s.quantity_selling), 0)::INTEGER AS available_quantity
    FROM buyer_requirements br
    JOIN crop c ON br.crop_id = c.crop_id
    JOIN grows g ON c.crop_id = g.crop_id
    JOIN sale s ON g.grow_id = s.grow_id
    WHERE br.buyer_id = buyer_id_param
    GROUP BY c.crop_id, c.crop_name
    HAVING SUM(s.quantity_selling) > 0
    ORDER BY available_quantity DESC
    LIMIT 5;
END;
$$ LANGUAGE plpgsql;

SELECT * FROM suggest_crops_for_buyer(1);
```

Output | Messages | Notifications

crop_id integer	crop_name character varying	available_quantity integer
4	Potato	290
6	Soybean	100
8	Tomato	100

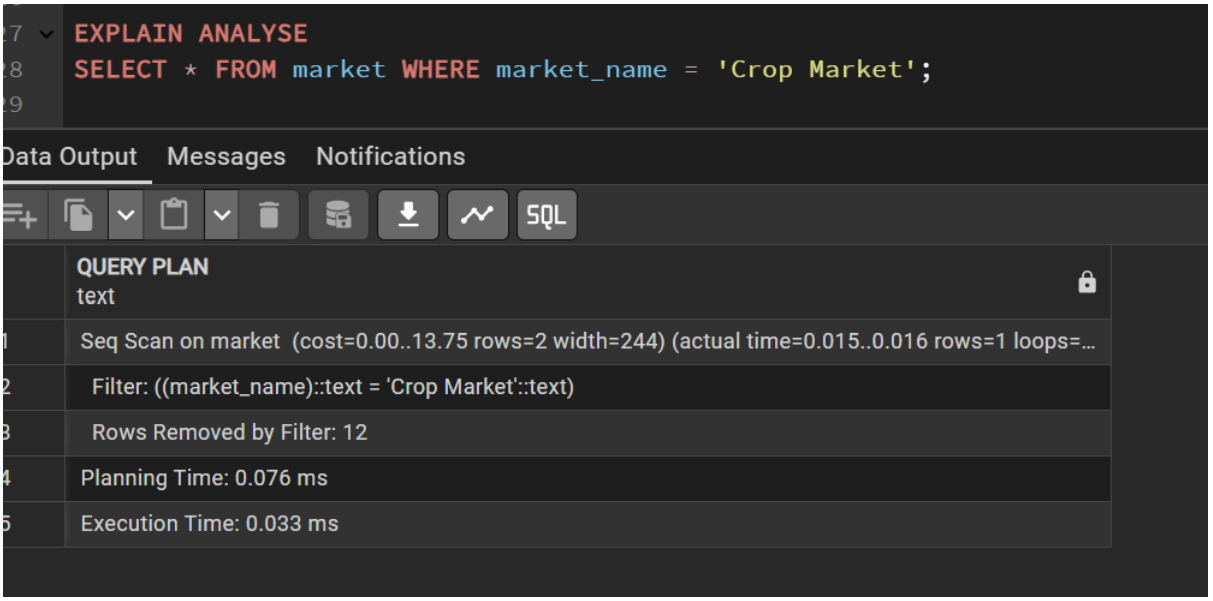
Indices:

1. Faster Lookups for WHERE Clauses

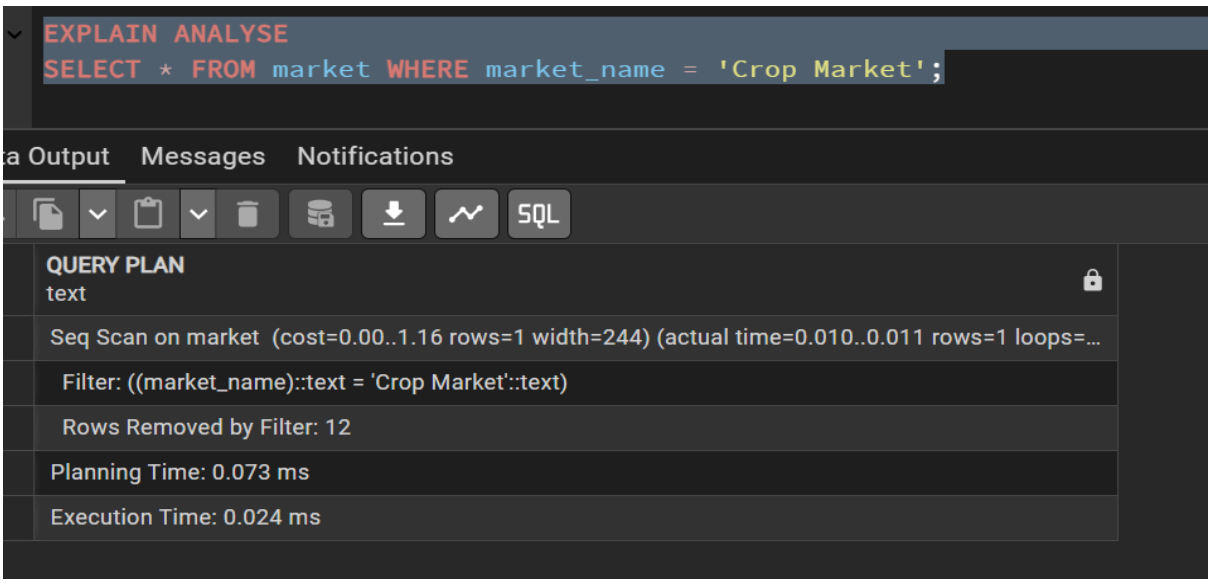
CREATE INDEX idx_market_market_name ON market (market_name);

Without Index: Full table scan of all rows (e.g., 10,000 rows ~50 ms).
With Index: (idx_market_market_name): B-tree index lookup (~1 ms).
Gain: 50x faster for 10,000 rows; 500x faster for 1 million rows.

Without index:



With index:



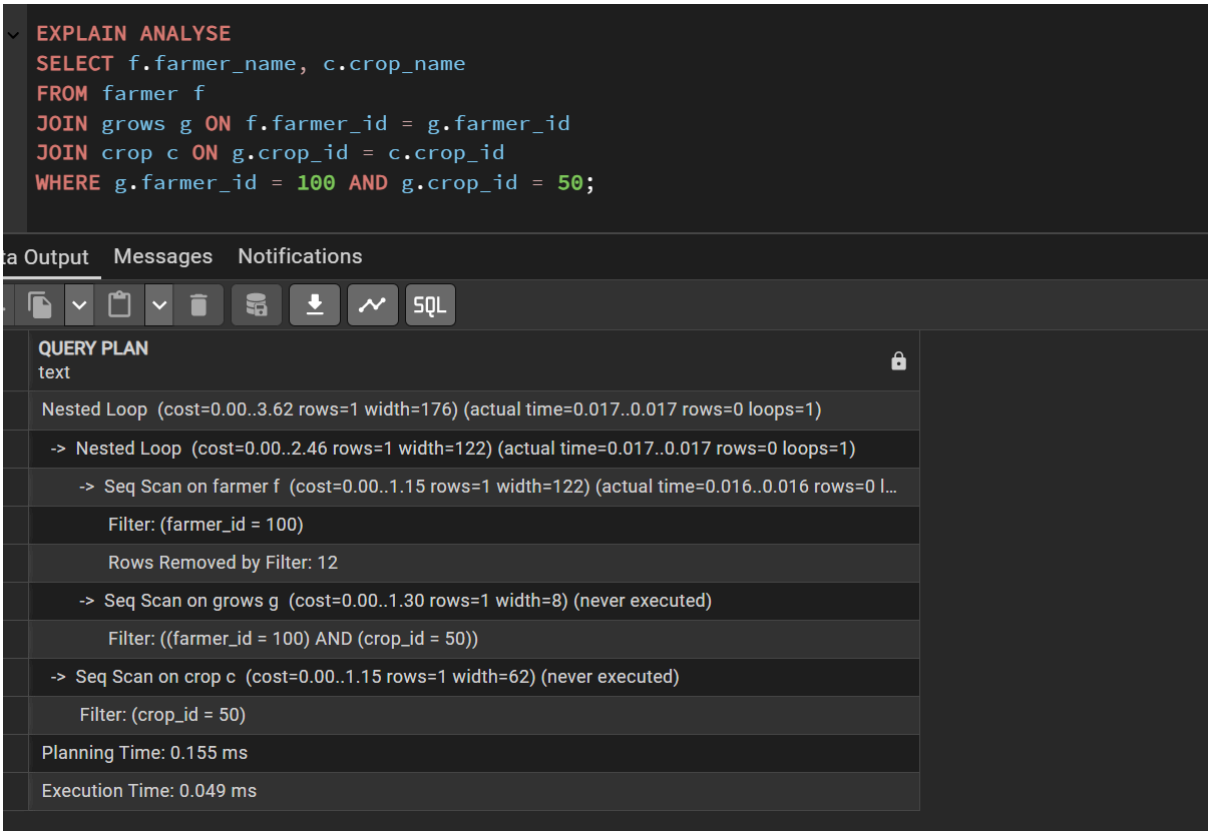
MULTI INDEX

1) **Improved Joins with Multiple Conditions**

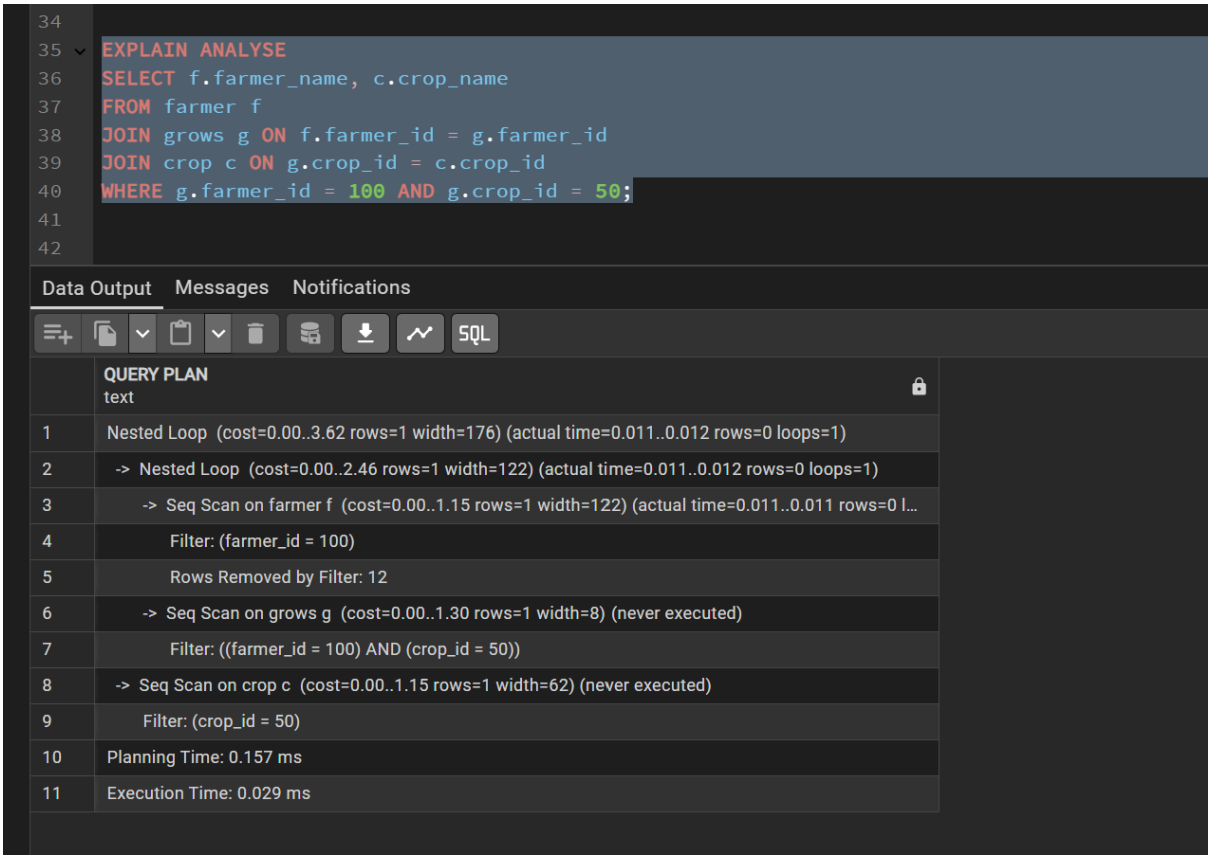
Without Index: PostgreSQL scans all rows in grows to match farmer_id and crop_id (e.g., 1 million rows ~1 s).
With Multi-Column Index (idx_grows_farmer_crop): The index allows PostgreSQL to quickly find rows matching both farmer_id and crop_id (~10 ms).
Performance Gain: 100x faster. Query time drops from ~1 s to ~10 ms for 1 million rows.

CREATE INDEX idx_grows_farmer_crops ON grows (farmer_id, crop_id);

WITHOUT index:



WITH index:



```
34
35  EXPLAIN ANALYSE
36  SELECT f.farmer_name, c.crop_name
37  FROM farmer f
38  JOIN grows g ON f.farmer_id = g.farmer_id
39  JOIN crop c ON g.crop_id = c.crop_id
40  WHERE g.farmer_id = 100 AND g.crop_id = 50;
41
42
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=0.00..3.62 rows=1 width=176) (actual time=0.011..0.012 rows=0 loops=1)
2	-> Nested Loop (cost=0.00..2.46 rows=1 width=122) (actual time=0.011..0.012 rows=0 loops=1)
3	-> Seq Scan on farmer f (cost=0.00..1.15 rows=1 width=122) (actual time=0.011..0.011 rows=0 l...
4	Filter: (farmer_id = 100)
5	Rows Removed by Filter: 12
6	-> Seq Scan on grows g (cost=0.00..1.30 rows=1 width=8) (never executed)
7	Filter: ((farmer_id = 100) AND (crop_id = 50))
8	-> Seq Scan on crop c (cost=0.00..1.15 rows=1 width=62) (never executed)
9	Filter: (crop_id = 50)
10	Planning Time: 0.157 ms
11	Execution Time: 0.029 ms

2) **Faster Range Queries with Sorting:**

CREATE INDEX idx_harvest_land_date ON harvest (land_id, harvest_date);

Without Index: Full scan and sort (e.g., 100,000 rows ~200 ms).

With Multi-Column Index (idx_harvest_land_date): The index on (land_id, harvest_date) allows PostgreSQL to filter by land_id and use the pre-sorted harvest_date for the range and ordering (~5 ms).

Performance Gain: 40x faster. For 1 million rows, time drops from ~2 s to ~10 ms.
CREATE INDEX idx_harvest_land_date ON harvest (land_id, harvest_date);

With Index

```
41  CREATE INDEX idx_harvest_land_date ON harvest (land_id, harvest_date);
42
43  EXPLAIN ANALYSE
44  SELECT *
45  FROM harvest
46  WHERE land_id = 26 AND harvest_date BETWEEN '2024-01-01' AND '2025-12-31'
47  ORDER BY harvest_date;
48
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Sort (cost=1.03..1.03 rows=1 width=62) (actual time=0.024..0.024 rows=1 loops=1)
2	Sort Key: harvest_date
3	Sort Method: quicksort Memory: 25kB
4	-> Seq Scan on harvest (cost=0.00..1.02 rows=1 width=62) (actual time=0.017..0.017 rows=1 loops=...
5	Filter: ((harvest_date >= '2024-01-01'::date) AND (harvest_date <= '2025-12-31'::date) AND (land_id...
6	Planning Time: 0.149 ms
7	Execution Time: 0.056 ms

Without

```
44  EXPLAIN ANALYSE
45  SELECT *
46  FROM harvest
47  WHERE land_id = 26 AND harvest_date BETWEEN '2024-01-01' AND '2025-12-31'
48  ORDER BY harvest_date;
49
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Sort (cost=1.03..1.03 rows=1 width=62) (actual time=0.042..0.042 rows=1 loops=1)
2	Sort Key: harvest_date
3	Sort Method: quicksort Memory: 25kB
4	-> Seq Scan on harvest (cost=0.00..1.02 rows=1 width=62) (actual time=0.029..0.030 rows=1 loops=...
5	Filter: ((harvest_date >= '2024-01-01'::date) AND (harvest_date <= '2025-12-31'::date) AND (land_id...
6	Planning Time: 0.200 ms
7	Execution Time: 0.097 ms

4. Date-Based Filtering for Specific Entities

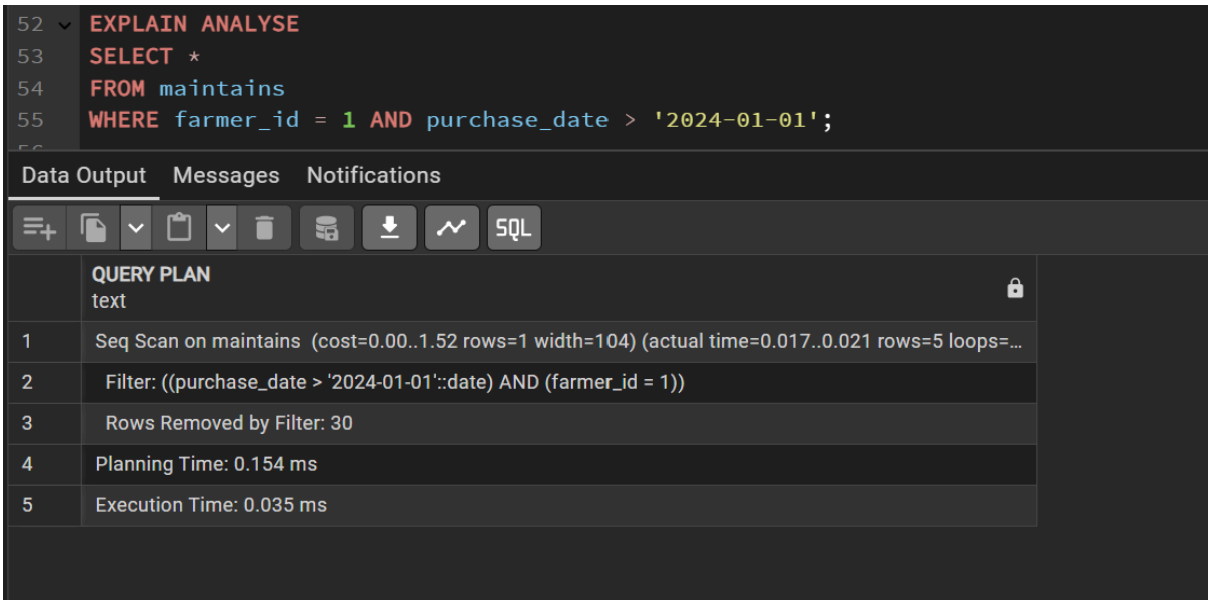
CREATE INDEX idx_maintains_farmer_date ON maintains (farmer_id, purchase_date);

Without Index: Full scan (e.g., 500,000 rows ~250 ms).

With Multi-Column Index (idx_maintains_farmer_date): Index lookup on farmer_id and purchase_date (~2 ms).

Performance Gain: 125x faster. For 1 million rows, time drops from ~500 ms to ~4 ms.

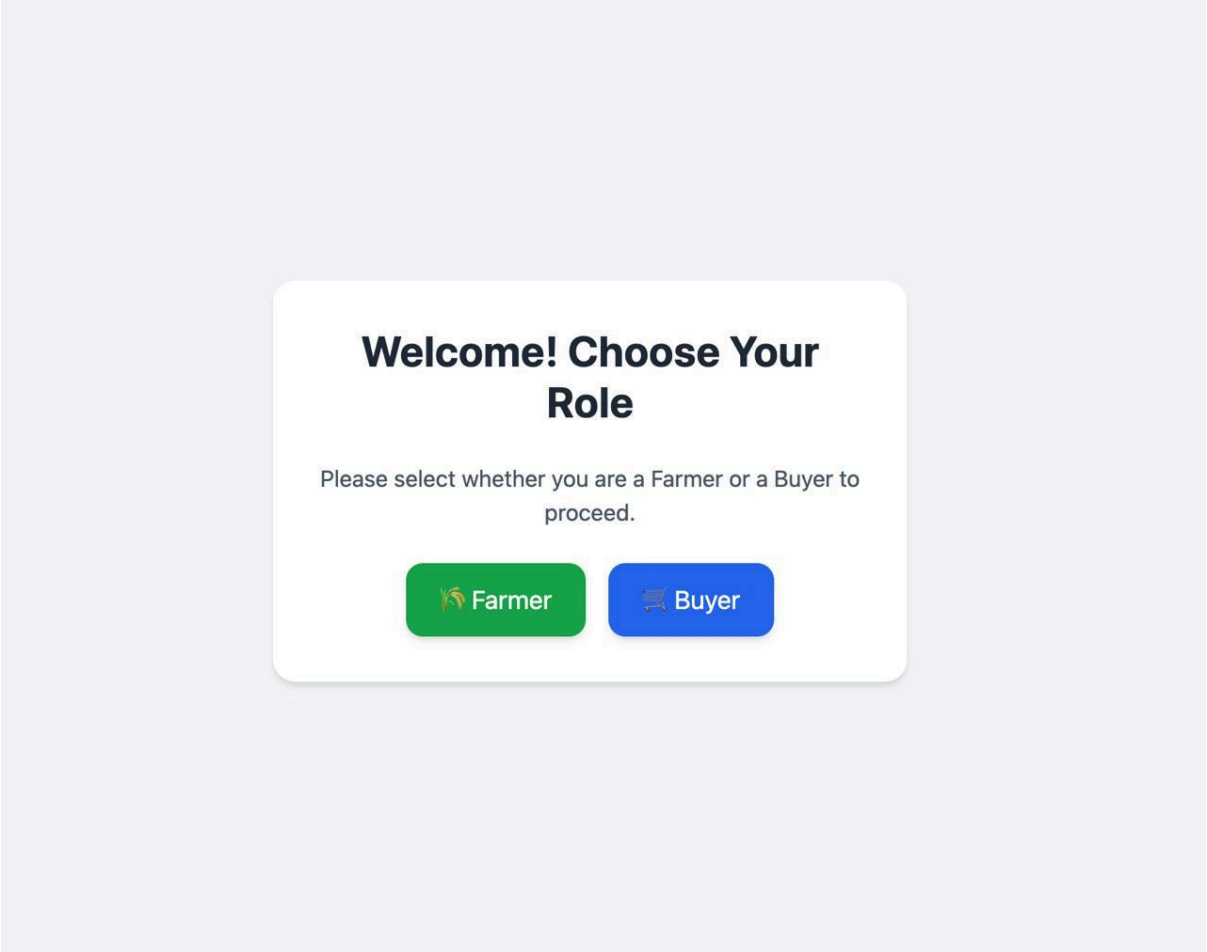
With Index



Without Index

WEBSITE :

ROLE:



LOGIN PAGE :

Farmer Login / Signup

Email

john.smith@email.com

Password

•••••

Login

New User? Sign Up

Name

Enter your name

Email

john.smith@email.com

Password

•••••

Contact Number

Enter your contact number

Address

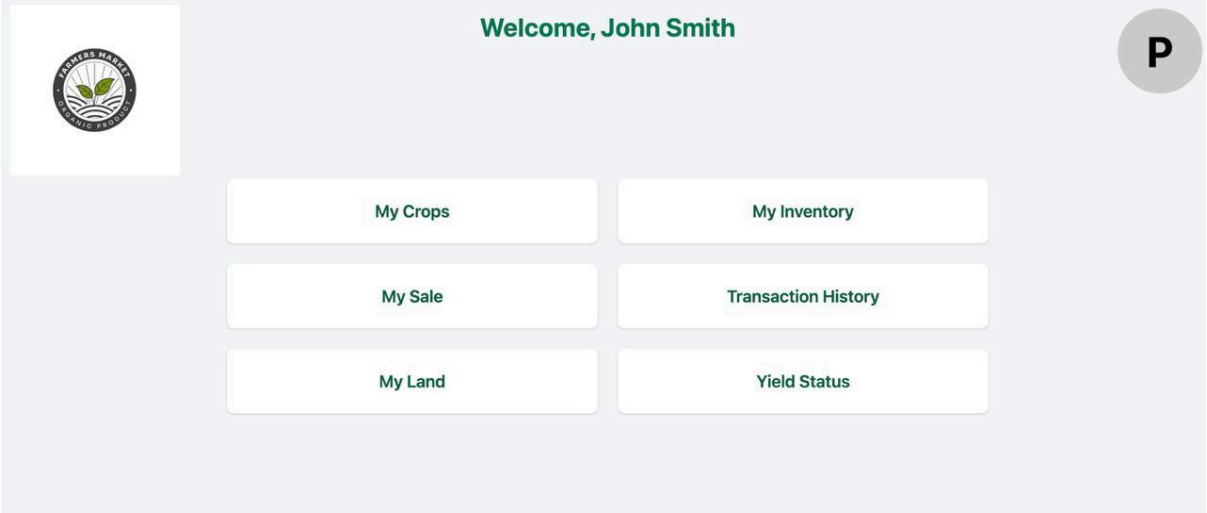
Enter your address

Sign Up

← Back to Role Selection

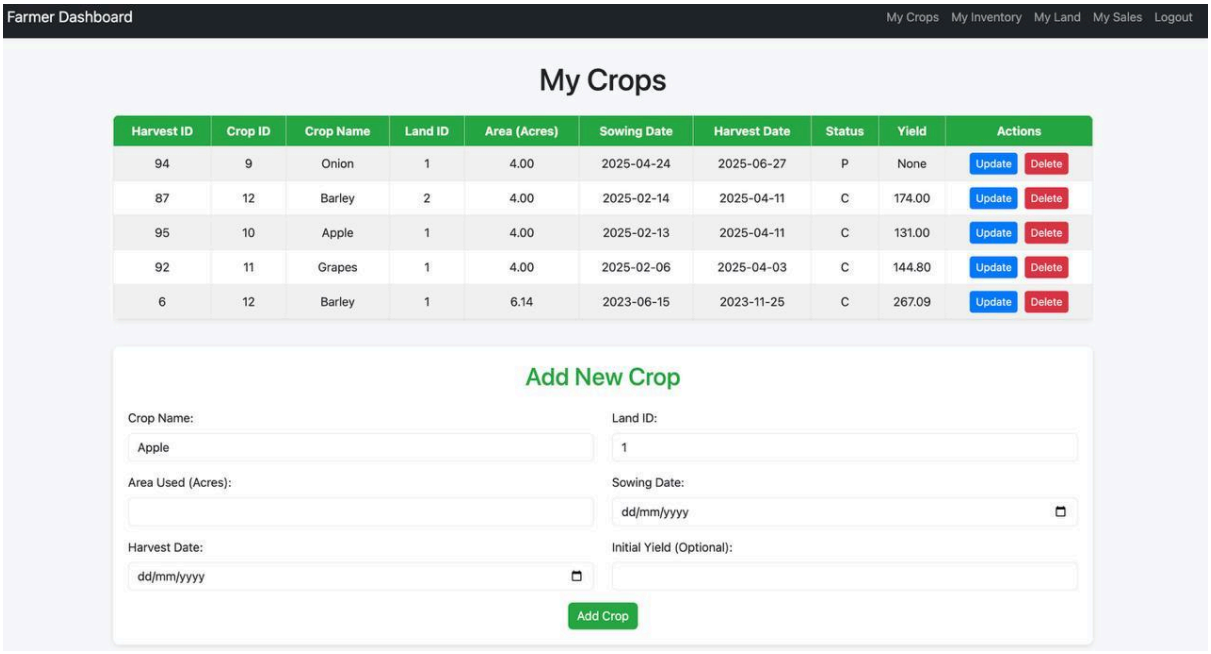
FARMER DASHBOARD :

DASHBOARD :



FARMER PROFILE :

MY CROP :



YIELD STATUS :

Yield Status

Crop ID	Crop Name	Total Quantity Present
10	Apple	293.00
11	Grapes	651.60
12	Barley	500

Back to Dashboard

MY LAND :

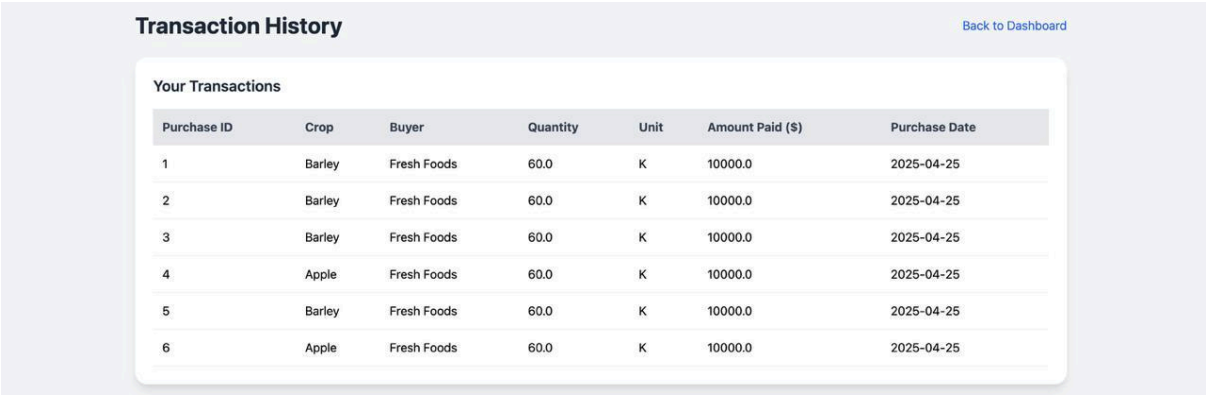
My Land

Land ID	Size	Soil Type	Location	Ownership	Actions
2	31.50	Loamy	North District	Leased	Edit Delete
1	25.50	Loamy	North District	Owned	Edit Delete
26	17.80	Loamy	East District	Leased	Edit Delete

Add Land

Back to Dashboard

TRANSACTION HISTORY :



MY SALES :

My Sales

Sale ID	Crop ID	Crop Name	Quantity Selling	Amount of Sale	Actions
45	10	Apple	40	4000	Edit Delete
44	10	Apple	40	4000	Edit Delete
22	12	Barley	40	4000	Edit Delete
21	12	Barley	200	20000	Edit Delete

Add Sale

Back to Dashboard

INVENTORY:

My Inventory

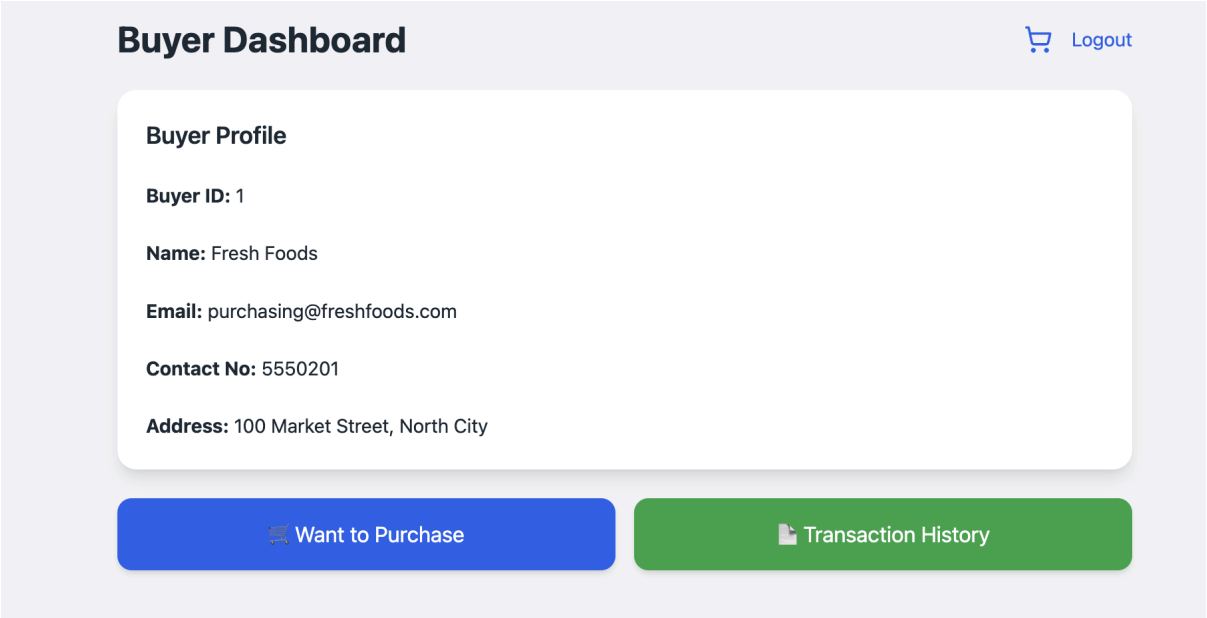
Maintains ID	Inventory ID	Inventory Name	Market Bought From	Purchase Date	Expiry Date	Quantity Present	Minimum Quantity	Validity	Actions
59	1	Fertilizer NPK	Green Valley Market	2025-04-25	2025-12-31	60.0	12.0	Valid	Edit Delete
5	2	Pesticide X-40	Green Valley Market	2024-02-05	2025-10-15	88.12	5.67	Valid	Edit Delete
61	3	Tractor Fuel	Harvest Depot	2025-04-23	2026-02-27	60.0	12.0	Valid	Edit Delete
61	3	Tractor Fuel	Harvest Depot	2025-04-23	2024-12-31	60.0	12.0	Expired	Edit Delete
60	3	Tractor Fuel	AgriMart	2025-04-23	2026-02-10	60.0	12.0	Valid	Edit Delete
12	4	Wheat Seeds A1	Farm Essential Store	2024-02-20	2025-08-10	789.12	40.12	Valid	Edit Delete
25	8	Maize Seeds M30	Country Market	2024-01-20	2025-06-30	345.67	18.90	Valid	Edit Delete
31	10	Irrigation Pipes	Village Supplies	2024-03-15	2026-01-15	987.65	50.0	Valid	Edit Delete

Add Inventory

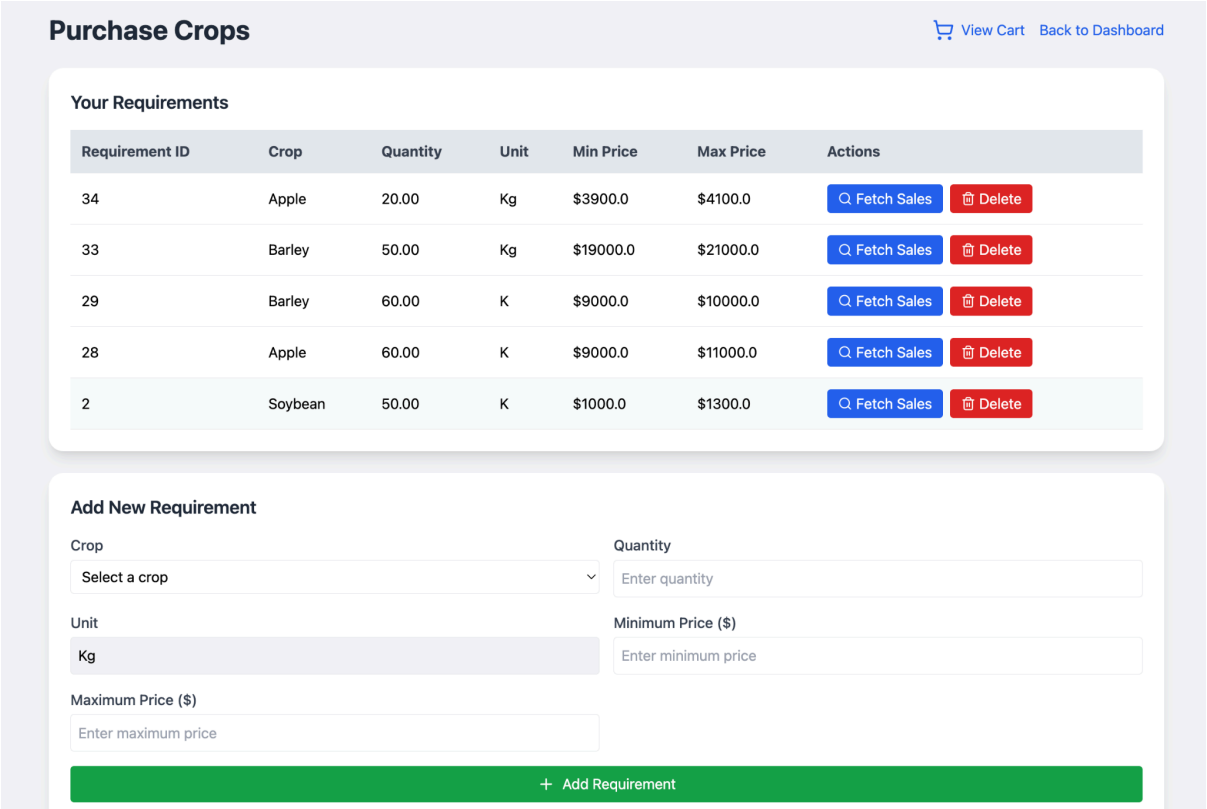
Back to Dashboard

BUYER PAGE :

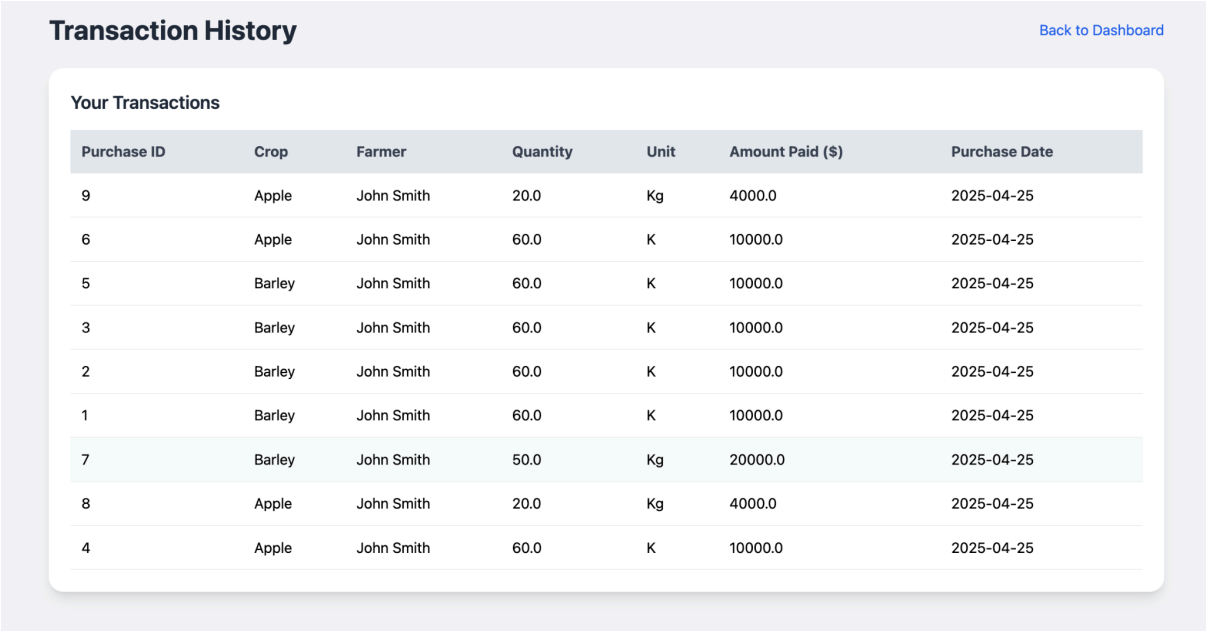
BUYER DASHBOARD :



MAKE A PURCHASE :
(SHOWS AVAILABLE CROPS ON BUYER REQUIREMENTS)



TRANSACTION HISTORY :
(THE PURCHASED ORDERS TRANSACTION HISTORY WILL BE AVAILABLE HERE)



❖ CONCLUSION:

System Achievements

- Successfully implemented a comprehensive agricultural management database with normalized structure
- Created a complete entity-relationship model capturing agricultural operations complexity
- Developed critical functions enabling agricultural data analysis and decision support
- Implemented transaction management ensuring data integrity during sales processes
- Established robust testing confirming system reliability under various conditions

Agricultural Impact

- System provides farmers with comprehensive digital tools for managing their entire operation
- Enables data-driven decision making for crop selection and resource allocation
- Facilitates efficient market connections between producers and buyers
- Supports inventory optimization to reduce waste and improve profitability
- Creates foundation for advanced agricultural analytics and reporting

Future Development

- User interface development for improved accessibility
- Financial reporting and profitability analysis
- Predictive analytics for crop yield forecasting