University of British Columbia, Vancouver
Department of Computer Science

CPSC 304 Project Cover Page

Milestone #4

Date: November 28th, 2025

Group Number: 31

Name	Student Number	CS Alias (Userid)	Preferred E-mail Address
Cameron Jim	64991375	c5x9i	cjim02@student.ubc.ca
Ryan Vu	93118289	x7m7j	ryanvu@student.ubc.ca
Wit Lin	46645636	k6g0d	wlin18@student.ubc.ca

By typing our names and student numbers in the above table, we certify that the work in the attached assignment was performed solely by those whose names and student IDs are included above. (In the case of Project Milestone 0, the main purpose of this page is for you to let us know your e-mail address, and then let us assign you to a TA for your project supervisor.)

In addition, we indicate that we are fully aware of the rules and consequences of plagiarism, as set forth by the Department of Computer Science and the University of British Columbia

Repository Link: github.students.cs.ubc.ca/CPSC304-2025W-T1/team_31

SQL Script Reference:
https://github.students.cs.ubc.ca/CPSC304-2025W-T1/team_31/blob/main/project.sql

Project Description/Accomplishments:
Our project is based in the sustainable agriculture/farm management space. The system will help farms keep organized records about the crops they grow and the certifications their farm has (ie. organic, non-GMO, etc.). Data is seasonal and can change over time, thus a centralized platform will help farmers review yearly crop records. Users are able to input information about their farms, including the field it is grown on, the types of crops, records of field conditions, and other miscellaneous things used like pesticides and records of irrigation. They are also able to update and delete their farm information, select specific fields based on attributes like area, fieldID and farmID, project specific columns from the ContainsField relation, look at which crop belongs to which farm, find the average irrigation used per field and the healthy fields, return the fieldID with the highest moisture and return fieldID that uses all pesticides.

The final schema made changes that were recommended based on the Milestone 2 rubric:

- The weak identity PK is now (CropID, TotalYield) since we had the incorrect PK
- FarmID was removed from the GrowsCrop relation (no relation to GrowsCrop)
- Area was removed from the SoilsRecords relation (since area was not a PK)

The changes for the latter two were made due to unnecessary information being in the table

SQL Queries:

#1 INSERT

Location: appService.js Line 618

```
async function insertFarm(farmID, name, location, farmerID) {
    // Validation with error handling
    if (!validate.isPositiveInteger(farmID)) {
        return { success: false, message: "Farm ID must be a positive integer." };
    }
    if (!validate.isValidString(name, 20)) {
        return { success: false, message: "Farm name must be between 1-20 characters." };
    }
    if (!validate.isValidString(location, 20)) {
        return { success: false, message: "Location must be between 1-20 characters." };
    }
    if (!validate.isPositiveInteger(farmerID)) {
        return { success: false, message: "Farmer ID must be a positive integer." };
    }

    return await withOracleDB(async (connection) => {
        try {
            const result = await connection.execute(
                `INSERT INTO OwnsFarm VALUES (:farmID, :name, :location, :farmerID)`,
                [farmID, name, location, farmerID],
                { autoCommit: true }
            );
            if (result.rowsAffected && result.rowsAffected > 0) {
                return { success: true, message: "Farm added successfully!" };
            }
            return { success: false, message: "Failed to add farm." };
        } catch (err) {
            console.error("Insert farm error:", err);

            // Handle specific Oracle errors
            if (err.errorNum === 1) {
                return { success: false, message: "Farm ID already exists. Please use a different ID." };
            } else if (err.errorNum === 2291) {
                return { success: false, message: "Farmer ID not found. Please enter a valid Farmer ID." };
            } else if (err.errorNum === 1400) {
                return { success: false, message: "Missing required field. All fields are required." };
            } else if (err.errorNum === 12899) {
                return { success: false, message: "Input too long. Please shorten your entries." };
            }

            return { success: false, message: "Database error: " + err.message };
        }
    }).catch((err) => {
        console.error("Connection error:", err);
        return { success: false, message: "Unable to connect to database." };
    });
}
```

#2 UPDATE

Location: appService.js Line 989

```
// Allows transfer of farm to new farmer, or change name/location of farm
async function updateFarmInfo(farmID, farmName, location, farmerID) {
    const updates = [];
    const values = {};

    values.farmID = farmID;

    if (farmName?.trim()) {
        updates.push("Name = :farmName");
        values.farmName = farmName;
    }

    if (location?.trim()) {
        updates.push("Location = :location");
        values.location = location;
    }

    if (farmerID?.trim()) {
        updates.push("FarmerID = :farmerID");
        values.farmerID = farmerID;
    }

    // Empty fields
    if (updates.length === 0) {
        return { success: false, message: "Update fields are all blank. Please check and try again." };
    }

    const sql = `UPDATE OwnsFarm
                 SET ${updates.join(", ")}
                 WHERE FarmID = :farmID`;

    return await withOracleDB(async (connection) => {
        try {
            const result = await connection.execute(
                sql,values, { autoCommit: true }
            );

            if (result.rowsAffected === 0) {
                return { success: false, message: "No farm found with this FarmID. Please check and try again." };
            }

            return { success: true, message: "Update successful!" };
        } catch (err) {
            // foreign key violation (does not exist)
            if (err.errorNum === 2291) {
                return { success: false, message: "farmerID not recognized. Please provide a valid existing farmerID." };
            }

            // log error and return generic err message
            console.error("Database error:", err);
            return { success: false, message: "Update failed due to an unexpected error"};
        }
    }).catch((err) => {
        console.error("Database connection failed:", err);
        return { success: false, message: "Update failed due to an unexpected error" };
    });
}
```

#3 DELETE

Location: appService.js Line 1061

```
// Delete farms
async function deleteFarmsInfo(farmID) {
    return await withOracleDB(async (connection) => {
        try {
            const result = await connection.execute(
                'DELETE FROM OwnsFarm WHERE FarmID = :farmID', [farmID], { autoCommit: true });

            if (result.rowsAffected === 0) {
                return { success: false, message: "No farm found with this FarmID. Please check and try again." };
            }

            return { success: true, message: "Cascade Delete successful!" };
        } catch (err) {
            console.error("Database error:", err);
            return { success: false, message: "Delete failed due to an unexpected error"};
        }
    }).catch((err) => {
        console.error("Database error:", err);
        return { success: false, message: "Delete failed due to an unexpected error"};
    })
};
```

#4 SELECTION

Location: appService.js Line 1195

```
async function selectFields(filter) {
    const validAttributes = ["FieldID", "FarmID", "Area"];
    const validOperators = ["=", "<>", ">", "<", ">=", "<="];
    const validConnectors = ["AND", "OR"];

    let whereParts = [];
    let binds = {};

    filter.conditions.forEach((cond, idx) => {
        const attr = cond.attribute;
        const op = cond.operator;
        const val = cond.value;
        const join = cond.connector;

        if (!validAttributes.includes(attr)) return;
        if (!validOperators.includes(op)) return;
        if (idx > 0 && !validConnectors.includes(join)) return;

        const bindKey = `v${idx}`;
        binds[bindKey] = val;

        const clause = `${attr} ${op} :${bindKey}`;

        if (idx === 0) {
            whereParts.push(clause);
        } else {
            whereParts.push(`${join} ${clause}`);
        }
    });

    // Select the whole table if no restrictions apply
    if (whereParts.length === 0) {
        const sql = `SELECT FieldID, FarmID, Area FROM ContainsField`;
        return await withOracleDB(async (connection) => {
            try {
                const result = await connection.execute(sql);
                return {
                    success: true,
                    message: "Selection successful!",
                    data: result.rows
                };
            } catch (err) {
                console.error("Selection SQL error:", err);
                return { success: false, message: "Selection failed due to an unexpected error." };
            }
        });
    }

    const where = "WHERE " + whereParts.join(" ");
    const sql = `SELECT FieldID, FarmID, Area
                FROM ContainsField
                ${where}
                `;

    return await withOracleDB(async (connection) => {
        try {
            const result = await connection.execute(sql, binds);
            return {
                success: true,
                message: 'Selection sucessful!',
                data: result.rows
            }
        } catch (err) {
            console.error("Database error:", err);
            return { success: false, message: "Selection failed due to an unexpected error." };
        }
    }).catch((err) => {
        console.error("Database error:", err);
        return { success: false, message: "Selection failed due to an unexpected error." };
    });
}
```

#5 Projection

Location: appService.js Line 1137

```
async function getFields(filter) {
    const fieldFields = ["FieldID", "FarmID", "Area"];

    const selectCols = [];

    if (!filter) {
        return {
            success: false,
            message: "No columns selected. Please choose at least one valid field.",
            data: []
        };
    }

    if (filter.display) {
        const cols = filter.display.split(",");

        cols.forEach(col => {
            // ContainsField table
            if (col === "FarmID")       selectCols.push("fd.FarmID AS FarmID");
            if (col === "FieldID")      selectCols.push("fd.FieldID AS FieldID");
            if (col === "Area")         selectCols.push("fd.Area AS Area");
        });
    }

    // Error handling, 0 features chosen send error
    if (selectCols.length === 0) {
        return {
            success: false,
            message: "None of the selected columns exist. Please choose valid fields.",
            data: []
        };
    }

    const select = `SELECT ${selectCols.join(", ")}`;
    const from = `FROM ContainsField fd
                `;

    const query = `${select} ${from}`;

    return await withOracleDB(async (connection) => {
        try {
            const result = await connection.execute(query);
            return {
                success: true,
                message: 'Projection sucessful!',
                data: result.rows
            }
        } catch (err) {
            console.error("Database error:", err);
            return { success: false, message: "Projection failed due to an unexpected error." };
        }
    }).catch((err) => {
        console.error("Database error:", err);
        return { success: false, message: "Projection failed due to an unexpected error." };
    });
}
```

#6 Join

Location: appService.js Line 1086

```
// Joins farms with the crops they grow
async function joinFarmCrop(farmID) {
    // Includes contact info and name even if farm has no crops or fields
    const sql = `SELECT
                    f.FarmID,
                    cin.Name AS FarmerName,
                    cin.ContactInfo,
                    ct.Name AS CropName
                FROM OwnsFarm f
                JOIN Farmer fr ON f.FarmerID = fr.FarmerID
                JOIN ContactInfoName cin ON fr.ContactInfo = cin.ContactInfo
                LEFT JOIN ContainsField fld ON f.farmID = fld.FarmID
                LEFT JOIN GrowsCrop gc ON fld.FieldID = gc.FieldID
                LEFT JOIN CropType ct ON gc.Name = ct.Name
                WHERE f.FarmID = :farmID
                ORDER BY ct.Name
                `

    return await withOracleDB(async (connection) => {
        try {
            const result = await connection.execute(
                sql,
                { farmID }
            );

            if (result.rows.length == 0) {
                return {
                    success: false,
                    message: "No farm found for this FarmID. Please check and try again.",
                    data: null
                };
            }

            return {
                success: true,
                message: "Search successful!",
                data: result.rows
            };
        } catch (err) {
            console.error("Database error:", err);
            return { success: false, message: "Query failed due to an unexpected error." };
        }
    }).catch((err) => {
        console.error("Database connection failed", err);
        return { success: false, message: "Query failed due to an unexpected error." };
    });
}
```

#7 GROUP BY
Location: appService.js Line 1286
Purpose: This query finds the average irrigated volume for each field. The grouping list is identified by farm name and fieldID, and the average volume is calculated.

Copy of SQL Query (does not have the ending semicolon):

```
SELECT i.FieldID, o.Name, AVG(i.Volume) as AvgVolume
        FROM IrrigationRecords i
        JOIN ContainsField f ON f.FieldID = i.FieldID
        JOIN OwnsFarm o ON o.FarmID = f.FarmID
        GROUP BY o.Name, i.FieldID
        ORDER BY i.FieldID ASC
```

```
// find the average volume for each field
async function fetchAverageVolume() {
    const sql = `SELECT i.FieldID, o.Name, AVG(i.Volume) as AvgVolume
                FROM IrrigationRecords i
                JOIN ContainsField f ON f.FieldID = i.FieldID
                JOIN OwnsFarm o ON o.FarmID = f.FarmID
                GROUP BY o.Name, i.FieldID
                ORDER BY i.FieldID ASC
                `;

    return await withOracleDB(async (connection) => {
        try {
            const result = await connection.execute(sql);

            return {
                success: true,
                message: "Search successful!",
                data: result.rows
            };
        } catch (err) {
            console.error("Database error:", err);
            return { success: false, message: "Query failed due to an unexpected error." };
        }
    }).catch((err) => {
        console.error("Database connection failed", err);
        return { success: false, message: "Query failed due to an unexpected error." };
    });
}
```

#8 HAVING
Location: appService.js Line 1297
Purpose: Returns all fields such that the average pH for each recording of the field is healthy (between 5.5 and 7.5). The having ensures that tuples are kept only if the field's (fieldID) average pH is healthy.

Copy of SQL Query (does not have the ending semicolon):

```
SELECT s.FieldID, o.Name, AVG(s.pH) as AvgpH
        FROM SoilRecords s
        JOIN ContainsField f ON f.FieldID = s.FieldID
        JOIN OwnsFarm o ON o.FarmID = f.FarmID
        GROUP BY o.Name, s.FieldID
        HAVING AVG(s.pH) BETWEEN 5.5 AND 7.5
        ORDER BY s.FieldID ASC
```

```
async function fetchHealthyFields() {
    const sql = `SELECT s.FieldID, o.Name, AVG(s.pH) as AvgpH
                FROM SoilRecords s
                JOIN ContainsField f ON f.FieldID = s.FieldID
                JOIN OwnsFarm o ON o.FarmID = f.FarmID
                GROUP BY o.Name, s.FieldID
                HAVING AVG(s.pH) BETWEEN 5.5 AND 7.5
                ORDER BY s.FieldID ASC
                `;

    return await withOracleDB(async (connection) => {
        try {
            const result = await connection.execute(sql);

            return {
                success: true,
                message: "Search successful!",
                data: result.rows
            };
        } catch (err) {
            console.error("Database error:", err);
            return { success: false, message: "Query failed due to an unexpected error." };
        }
    }).catch((err) => {
        console.error("Database connection failed", err);
        return { success: false, message: "Query failed due to an unexpected error." };
    });
}
```

#9 Nested Aggregation
Location: appService.js Line 1328
Purpose: This query returns the fieldID with the highest moisture.

Copy of SQL Query (does not have the ending semicolon):

```
SELECT sr.FieldID
        FROM SoilRecords sr
        JOIN MoistureByChemistry mc
        ON sr.SampleDate = mc.SampleDate AND sr.pH = mc.pH
        GROUP BY sr.FieldID
        HAVING AVG(mc.moisture) >= ALL (
          SELECT AVG(mc2.moisture)
          FROM SoilRecords sr2
          JOIN MoistureByChemistry mc2
          ON sr2.SampleDate = mc2.SampleDate AND sr2.pH = mc2.pH
          GROUP BY sr2.FieldID
        )
```

```
async function fetchHighestMoistureField() {
    const sql = `SELECT sr.FieldID
                FROM SoilRecords sr
                JOIN MoistureByChemistry mc
                ON sr.SampleDate = mc.SampleDate AND sr.pH = mc.pH
                GROUP BY sr.FieldID
                HAVING AVG(mc.moisture) >= ALL (
                    SELECT AVG(mc2.moisture)
                    FROM SoilRecords sr2
                    JOIN MoistureByChemistry mc2
                    ON sr2.SampleDate = mc2.SampleDate AND sr2.pH = mc2.pH
                    GROUP BY sr2.FieldID
                )
                `
    return await withOracleDB(async (connection) => {
        try {
            const result = await connection.execute(sql);

            return {
                success: true,
                message: "Search successful!",
                data: result.rows
            };
        } catch (err) {
            console.error("Database error:", err);
            return { success: false, message: "Query failed due to an unexpected error." };
        }
    }).catch((err) => {
        console.error("Database connection failed", err);
        return { success: false, message: "Query failed due to an unexpected error." };
    });
}
```

#10 Division
Location: appService.js Line 1362
Purpose: This query returns all fieldID's that contain all pesticides. Fields are included if no pesticides are missing.

Copy of SQL Query (does not have the ending semicolon):

```
SELECT cf.FieldID
        FROM ContainsField cf
        WHERE NOT EXISTS (
          (SELECT p.pestID
          FROM Pesticide p)

          MINUS

          (SELECT t.PestID
          FROM GrowsCrop gc
          JOIN Treats t ON gc.CropID = t.CropID
          WHERE gc.FieldID = cf.FieldID
          )
        )
```

```
async function fetchFieldsAllPesticides() {
    const sql = `SELECT cf.FieldID
                FROM ContainsField cf
                WHERE NOT EXISTS (
                    (SELECT p.pestID
                    FROM Pesticide p)

                    MINUS

                    (SELECT t.PestID
                    FROM GrowsCrop gc
                    JOIN Treats t ON gc.CropID = t.CropID
                    WHERE gc.FieldID = cf.FieldID
                    )
                )
                `
    return await withOracleDB(async (connection) => {
        try {
            const result = await connection.execute(sql);

            return {
                success: true,
                message: "Search successful!",
                data: result.rows
            };
        } catch (err) {
            console.error("Database error:", err);
            return { success: false, message: "Query failed due to an unexpected error." };
        }
    }).catch((err) => {
        console.error("Database connection failed", err);
        return { success: false, message: "Query failed due to an unexpected error." };
    });
}
```

Schemas after Initialization of SQL Script:

```
CREATE TABLE ContactInfoName (
  ContactInfo  VARCHAR(80) PRIMARY KEY,
  Name         VARCHAR(20)  NOT NULL
);
```

ContactInfo	Name
john.doe@example.com	John Doe
jack.smith@example.com	Jack Smith
brian.smith@example.com	Brian Smith
cora.doe@example.com	Cora Doe
eve.miller@example.com	Eve Miller

```
CREATE TABLE Farmer (
  FarmerID     INT PRIMARY KEY,
  ContactInfo  VARCHAR(80) NOT NULL,
  CONSTRAINT fk_farmer_contact
    FOREIGN KEY (ContactInfo) REFERENCES
ContactInfoName(ContactInfo),
  CONSTRAINT uq_farmer_contact UNIQUE
(ContactInfo)
);
```

FarmerID	ContactInfo
1	john.doe@example.com
2	jack.smith@example.com
3	brian.smith@example.com
4	cora.doe@example.com
5	eve.miller@example.com

```
CREATE TABLE OwnsFarm (
  FarmID    INT PRIMARY KEY,
  Name      VARCHAR(20) NOT NULL,
  Location  VARCHAR(20) NOT NULL,
  FarmerID  INT NOT NULL,
  CONSTRAINT fk_farm_farmer
    FOREIGN KEY (FarmerID) REFERENCES
Farmer(FarmerID)
);
```

FarmID	Name	Location	FarmerID
101	Sunny Fields	British Columbia	1
102	Green Valley	Alberta	2
103	River Farm	Ontario	3
104	Highland Acres	British Columbia	4
105	Golden Harvest	Manitoba	5

```
CREATE TABLE ContainsField (
  FieldID  INT PRIMARY KEY,
  FarmID   INT NOT NULL,
  Area     INT,
  CONSTRAINT fk_field_farm
    FOREIGN KEY (FarmID) REFERENCES
OwnsFarm(FarmID)
);
```

FieldID	FarmID	Area
1001	101	25
1002	102	40
1003	103	30
1004	104	35
1005	105	25
1006	103	65

```
CREATE TABLE SeasonByPlantDate (
  PlantingDate DATE PRIMARY KEY,
  Season       VARCHAR(20) NOT NULL
);
```

PlantingDate	Season
2025-03-10	Spring
2025-04-15	Spring
2025-05-20	Summer
2025-07-01	Summer
2025-09-10	Fall
2025-10-10	Fall

```
CREATE TABLE CropType (
  Name         VARCHAR(60) PRIMARY KEY,
  PlantingDate  DATE NOT NULL,
  HarvestDate   DATE NOT NULL,
  CONSTRAINT fk_croptype_plantdate
    FOREIGN KEY (PlantingDate) REFERENCES
SeasonByPlantDate(PlantingDate)
);
```

Name	PlantingDate	HarvestDate
Wheat	2025-03-10	2025-07-20
Corn	2025-04-15	2025-09-05
Tomato	2025-05-20	2025-08-25
Lettuce	2025-07-01	2025-08-01
Apple	2025-09-10	2026-03-01
Banana	2025-10-10	2026-06-01

```
CREATE TABLE GrowsCrop (
  CropID   INT PRIMARY KEY,
  FieldID  INT NOT NULL,
  Name     VARCHAR(60) NOT NULL,
  CONSTRAINT fk_crop_field
    FOREIGN KEY (FieldID) REFERENCES
ContainsField(FieldID),
  CONSTRAINT fk_crop_name
    FOREIGN KEY (Name) REFERENCES
CropType(Name)
);
```

CropID	FieldID	Name
201	1001	Wheat
206	1001	Corn
207	1001	Tomato
208	1001	Banana
202	1002	Corn
203	1003	Tomato
204	1004	Lettuce
205	1005	Apple
209	1005	Banana
210	1003	Cucumber
211	1003	Basil
212	1003	Orange
213	1003	Kiwi
214	1003	Rice
215	1003	Oats

```
CREATE TABLE Grain (
  CropID        INT PRIMARY KEY,
  GlutenContent  DECIMAL(10,2),
  CONSTRAINT fk_grain_crop
    FOREIGN KEY (CropID) REFERENCES
GrowsCrop(CropID)
     ON DELETE CASCADE
);
```

CropID	GlutenContent
201	12.5
202	0
206	0
214	0
215	1

```
CREATE TABLE Vegetable (
  CropID   INT PRIMARY KEY,
  IsLeafy  NUMBER(1) NOT NULL,
  CONSTRAINT ck_veg_bool CHECK (IsLeafy IN
(0,1)),
  CONSTRAINT fk_veg_crop
    FOREIGN KEY (CropID) REFERENCES
GrowsCrop(CropID)
      ON DELETE CASCADE
);
```

CropID	IsLeafy
203	0
204	1
207	0
210	0
211	1

```
CREATE TABLE Fruit (
  CropID       INT PRIMARY KEY,
  SugarContent DECIMAL(10,2),
  CONSTRAINT fk_fruit_crop
    FOREIGN KEY (CropID) REFERENCES
GrowsCrop(CropID)
      ON DELETE CASCADE
);
```

CropID	SugarContent
205	14.2
208	12.3
209	16.8
212	15.6
213	12.8

```
CREATE TABLE CropYieldProduces (
  CropID       INT,
  Total_Yield   DECIMAL(10,2) NOT NULL,
  Health_Rating INT NOT NULL,
  PRIMARY KEY (CropID, Total_Yield),
  CONSTRAINT fk_yield_crop
    FOREIGN KEY (CropID) REFERENCES
GrowsCrop(CropID)
      ON DELETE CASCADE
);
```

CropID	Total_Yield	Health_Rating
201	5000	9
202	6500.5	8
203	2200.75	7
204	1800	9
205	8000.25	10

```
CREATE TABLE Pesticide (
  PestID  INT PRIMARY KEY,
  Name    VARCHAR(60)
);
```

PestID	Name
1	Mr Clean
2	Pest Killer
3	P-Cleaner Deluxe
4	Bye Bye Pests
5	Get Outta Here Pests
6	Pest Eliminator 3000

```
CREATE TABLE Treats (
  CropID  INT,
  PestID  INT,
  PRIMARY KEY (CropID, PestID),
  CONSTRAINT fk_treats_crop
    FOREIGN KEY (CropID) REFERENCES
GrowsCrop(CropID)
      ON DELETE CASCADE,
  CONSTRAINT fk_treats_pest
    FOREIGN KEY (PestID) REFERENCES
Pesticide(PestID)
      ON DELETE CASCADE
);
```

CropID	PestID
201	1
201	2
201	3
201	4
201	5
201	6
202	1
202	3
203	2
205	1
205	2
205	3
205	4
205	5
205	6

```
CREATE TABLE IrrigationRecords (
  IrrigID    INT PRIMARY KEY,
  FieldID    INT NOT NULL,
  EventDate  DATE,
  Volume     DECIMAL(10,2),
  CONSTRAINT fk_irrig_field
    FOREIGN KEY (FieldID) REFERENCES
ContainsField(FieldID)
);
```

IrrigID	FieldID	EventDate	Volume
1	1001	2023-03-01	1500
2	1001	2023-04-15	2000
3	1002	2023-05-10	1800
4	1002	2023-06-20	2200
5	1003	2023-07-05	1600
6	1003	2023-08-18	2100

```
CREATE TABLE MoistureByChemistry (
  SampleDate DATE,
  pH         DECIMAL(4,2),
  Moisture   DECIMAL(6,2) NOT NULL,
  PRIMARY KEY (SampleDate, pH)
);
```

SampleDate	pH	Moisture
2023-03-15	6.5	22.5
2023-04-20	7	18
2023-05-10	5.8	25
2023-06-25	6.2	20
2023-07-30	7.5	15
2023-08-15	6.9	19.5
2023-09-01	4	28
2023-09-15	4.5	24
2023-10-05	5.2	30
2023-10-20	6.8	27.5
2024-09-15	5.5	24
2024-10-05	5.7	30
2024-10-20	4.2	27.5

```
CREATE TABLE SoilRecords (
  SoilCondID  INT PRIMARY KEY,
  FieldID     INT NOT NULL,
  SampleDate  DATE NOT NULL,
  pH          DECIMAL(4,2) NOT NULL,
  CONSTRAINT fk_soil_field
    FOREIGN KEY (FieldID) REFERENCES
ContainsField(FieldID),
  CONSTRAINT fk_soil_rule
    FOREIGN KEY (SampleDate, pH)
REFERENCES
MoistureByChemistry(SampleDate, pH)
);
```

SoilCondID	FieldID	SampleDate	pH
1	1001	2023-03-15	6.5
2	1001	2023-04-20	7
3	1001	2023-10-05	5.2
4	1002	2023-05-10	5.8
5	1002	2023-06-25	6.2
6	1002	2023-10-20	6.8
7	1003	2023-07-30	7.5
8	1003	2023-08-15	6.9
9	1003	2023-09-01	4
10	1004	2023-09-15	4.5
11	1004	2023-10-05	5.2
12	1004	2023-10-20	6.8
13	1005	2023-09-01	4
14	1005	2023-09-15	4.5
15	1005	2023-10-05	5.2
16	1006	2024-09-15	5.5
17	1006	2024-10-05	5.7
18	1006	2024-10-20	4.2

```
CREATE TABLE AwardExpiry (
  AwardedDate DATE PRIMARY KEY,
  ExpiryDate  DATE NOT NULL
);
```

AwardedDate	ExpiryDate
2021-10-16	2023-10-16
2022-01-15	2024-01-15
2023-02-15	2025-02-15
2024-05-11	2026-05-11
2024-12-30	2026-12-30

```
CREATE TABLE Certification (
  CertID      INT PRIMARY KEY,
  Name        VARCHAR(80) NOT NULL,
  AwardedDate DATE NOT NULL,
  CONSTRAINT fk_cert_award
    FOREIGN KEY (AwardedDate) REFERENCES
AwardExpiry(AwardedDate)
);
```

CertID	Name	AwardedDate
1	Organic	2022-01-15
2	Winner	2023-02-15
3	BestFarm	2024-12-30
4	PrettiestFarm	2024-05-11
5	BestEnvironment	2021-10-16

```
CREATE TABLE Receives (
  FarmID  INT,
  CertID  INT,
  PRIMARY KEY (FarmID, CertID),
  CONSTRAINT fk_recv_farm
    FOREIGN KEY (FarmID) REFERENCES
OwnsFarm(FarmID),
  CONSTRAINT fk_recv_cert
    FOREIGN KEY (CertID) REFERENCES
Certification(CertID)
);
```

FarmID	CertID
101	1
101	2
102	1
103	3
104	2
105	1