Designing Sustainable Agriculture Databases: Relational and Document-Based Approaches

Student Number - 720017170

November 26, 2025

1 Introduction

This report describes the design, implementation, and analysis of a data system which stores UK agricultural sustainability initiatives. An efficient and reliable database is developed to store and analyse the performance of sustainability initiatives across UK farms.

The project discusses four key sections:

- **Relational Database Design:** A relational database design adhering to the third normal form (3NF), as well as an entity relationship diagram (ERD), supported by detailed design rationale.
- **Database Implementation:** An outline and `.sql` script, implemented in MySQL. The script is used to create and populate the database tables from raw, unnormalized data.
- **RESTful API Design & Implementation:** The design and implementation of a RESTful API that allows for data access and modification within the database, including all endpoints and their functions.
- **Alternative Design:** An alternative design using a document-based NoSQL database model and a comparison with a relational approach, considering the trade-offs in scalability, data integrity, flexibility, and performance.

2 Database Design

The database was designed to 3NF, where the key considerations were data integrity, reducing redundancy, and ensuring efficient data retrieval; all of which are important for designing production-level databases [1].

2.1 Design Rationale

The aim of the design was to remove all data redundancy and inconsistencies. This was achieved by ensuring all non-key attributes are fully dependent only on the primary key of that table with no transitive dependencies [1]. By adhering to 3NF, the design will eliminate update, insertion, and deletion errors, which in turn increasing consistency and accuracy [1].

2.2 Normalisation and Referential Integrity

The design process began by considering first and second normal forms (1NF, 2NF). The raw data was processed to ensure 1NF by guaranteeing that each row and column contains only a single value, and all duplicate groups were removed [1]. To achieve 2NF, each non-primary key attribute was made fully dependent on only the primary key of its table [1]. For example, attributes such as crop_name and resource_type were dependent only on their tables primary keys, not on any other records. Consequently, descriptive tables (Crop, Resource), which stored each value only once, were created to eliminate repeated data storage [1].

Several junction tables were created to implement 3NF, as all transitive dependencies must be removed [1]. For instance, the Farm_Crop table resolves the many-to-many relationship between farms and crops, ensuring temporal attributes (planting_date, harvest_date) that are specific to each farm's growing of a crop, are accurately recorded. The Crop_Resource table captures the relationship between crops and resources, storing when each resource was applied to a crop and in what quantity.

To enable the analysis of initiative impact on crop performance, Crop_Initiative was designed to link crops to their respective sustainability initiative. Also, the Farm_Initiative table consolidates the relationship between farms and initiatives, storing initiative-specific metrics for each farm (date_initiated, expected_impact, ev_score, water_source). This allows the same initiative to be tracked separately across different farms, with potentially different values per farm [1].

Labour hours are captured in the Labour table using the unique combination of farm, crop, resource, and initiative. The combination is necessary because labour requirements are dependent on which resources are applied to a specific crop on a farm that is using a particular initiative.

Finally, all junction tables use surrogate primary keys for efficient indexing, while foreign key constraints enforce referential integrity and prevent data inconsistency across the database.

2.3 Entity Relationship Diagram (ERD)

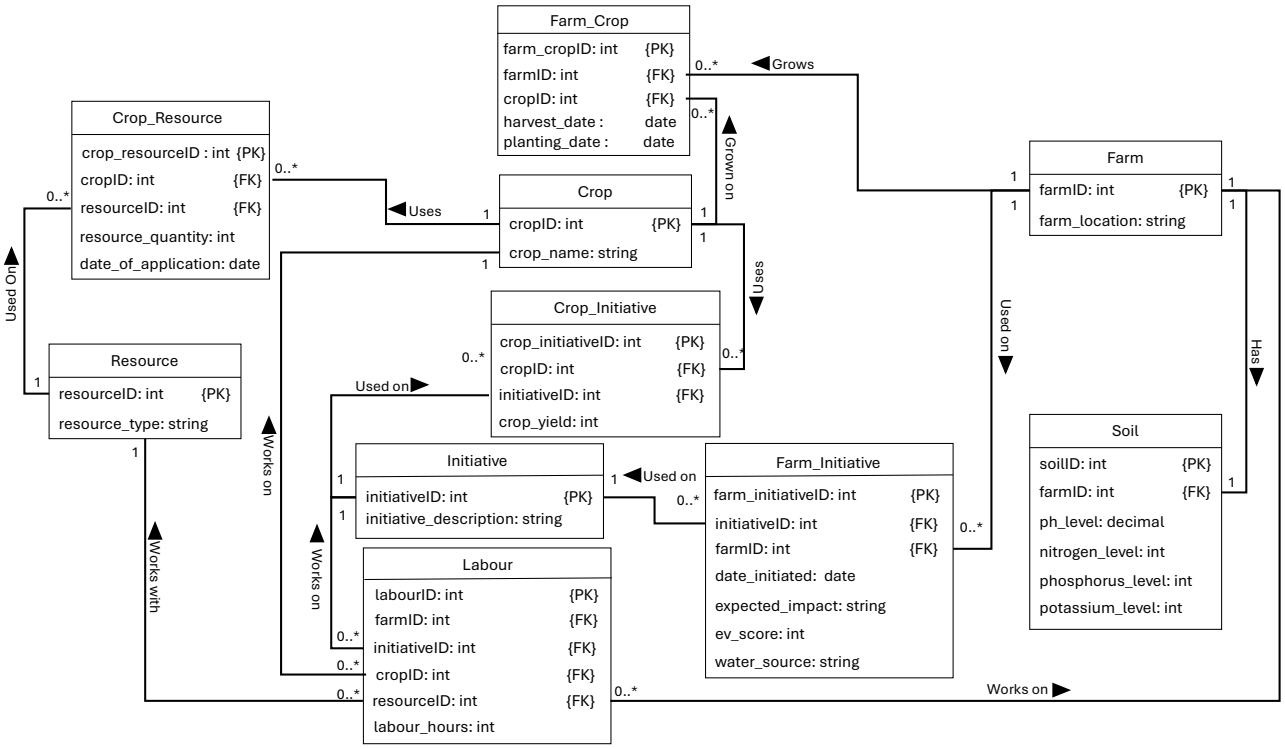

Figure 1: Normalised (3NF) Entity Relationship Diagram

3 SQL Implementation

Section 3 discusses the processes taken to develop the database within MySQL from the E-R diagram in Figure 1, with more detailed implementation and SQL provided in the accompanying `.sql` script.

3.1 Database Configuration

First, a new database environment was created within MySQL Workbench, making sure the database implementation is clean and reproducible. The process removes any previous version of the database in that environment. Following this, the database can be created using the `CREATE DATABASE` command. To allow file population from the CSV provided, local file loading is enabled. The specific set up guarantees that the database is reproducible from a blank baseline so that all future commands work within the environment.

3.2 Table Creation

After the database was created and configured, tables were created to store each record. Initially, descriptive tables were created to contain data that only related to Farms, Soils, Crops, Initiatives and Resources. Junction tables also were defined in order to facilitate many-to-many relationships as well as store instance specific attributes. These included Farm_Crop, Crop_Resource, Crop_Initiative, Farm_Initiative, and Labour. Each table was created with appropriate primary keys, foreign keys and constraints to ensure data integrity.

3.3 Data Loading and Cleaning

First, the raw data provided was loaded into a staging table using the MySQL `LOAD DATA INFILE` command. The staging table mirrors the structure of the raw CSV file, ensuring a clean import, as well as providing a location for data cleaning. Once within the staging table, certain columns required cleaning to ensure compatibility with the final database schema. Cleaning included standardising date formats, due to formatting inconsistencies resulting from loading.

After cleaning, the data was then inserted into the schema using `INSERT INTO ... SELECT` statements. These statements extract the relevant data from the staging table, and load it into the final schema. The cleaning process meant that all data in the final tables adhered to the defined structure and integrity constraints of the database schema. After population, the database was tested with various queries to validate the design and implementation, all shown at the end of the `.sql` script.

3.4 Cloud Extension

Beyond the core requirements, the database was set up in a cloud environment using Microsoft Azure's SQL Database service. This involved creating a new SQL schema within Azure using MSSQL. The existing `.sql` script was adapted to be compatible with Azure's SQL syntax and features. A Python script was developed to automate the data loading process into the Azure database, ensuring that the data could be efficiently and accurately populated in the cloud environment. The cloud implementation facilitates RESTful API access.

4 RESTful API Design

4.1 Design

To allow the database to be accessible from other applications, a RESTful API was designed to facilitate data access and modification. The API uses standard HTTP methods (GET, POST, PUT, PATCH, DELETE) which perform CRUD operations within the database [2].

The design utilises generic endpoints, shown in Table 1, able to access and manipulate data across all tables. For example, `GET /:table/:id` gives access to all data from any table by specifying the table name and an optional record ID, implementing request flexibility along with code reusability and simplicity during implementation. The generic approach does sacrifice some specificity and optimisation that more tailored endpoints may achieve. The `/query` endpoint permits custom SQL queries, providing flexibility to data analysts. Flexibility reduces constraints caused by predefined endpoints. Letting users enter custom SQL could introduce security risks if not properly managed; however, security considerations can be mitigated by restricting access to read-only queries and implementing access control.

In a production environment, additional tailored endpoints could be implemented for frequently accessed resources, such as `GET /farms/farmID/environmental-score` to retrieve sustainability metrics for specific farms, or `GET /crops/cropID/yield` to access crop performance data. These would optimise performance for common use cases while the generic endpoints provide flexibility for ad-hoc analysis.

All data is exchanged in JSON format, which is widely used and supported across various programming languages and platforms [2]. The design along with concise error codes and messages, makes the API user-friendly and accessible for developers interacting with the agricultural database.

Table 1: RESTful API Endpoint Definitions

Method	Endpoint	Parameters	Description	Response
GET	`/:table`	**Query:** `limit` (int), `offset` (int), `sort_by` (string), `order` (asc|desc)	Retrieves all records from specified table with pagination and sorting.	200 OK: JSON array
GET	`/:table/:id`	**Path:** `:id` (int) — primary key	Retrieves single record by primary key. Returns empty if not found.	200 OK: JSON object | 404 Not Found
POST	`/:table`	**Body:** JSON object matching table schema. Example for Farm: `{"farm_location": "string"}`	Creates new record with provided column-value pairs.	201 Created: JSON `{ok, rowsAffected}` | 400 Bad Request
PUT	`/:table/:id`	**Path:** `:id` (int) **Body:** Complete JSON object with all table fields	Full record replacement. All fields must be provided.	200 OK | 404 Not Found | 400 Bad Request
PATCH	`/:table/:id`	**Path:** `:id` (int) **Body:** Partial JSON object with only fields to update	Partial update. Only specified fields modified.	200 OK | 404 Not Found | 400 Bad Request
DELETE	`/:table/:id`	**Path:** `:id` (int)	Deletes record by primary key.	200 OK: `{ok, message}` | 404 Not Found
POST	`/query`	**Body:** `{"sql": "SELECT ... FROM ..."}` **Restriction:** SELECT-only queries permitted	Executes read-only SELECT queries, others blocked for security.	200 OK: JSON array | 400 Bad Request | 403 Forbidden

4.2 Implementation Extension

To extend the coursework and validate my design, I developed the RESTful API using Node.js, Express and MSSQL to access and query the cloud database. The API implements all the designed endpoints outlined in Section 4.1, and is hosted on Azure App Service, providing a scalable and reliable platform for deployment. The deployed API can be accessed at: `http://agriculture-api-jlg-g2gffahdgxfqc6b3.swedencentral-01.azurewebsites.net/` and the interface is available at: `https://joshlg18.github.io/Agricultural-Database/`.

5 Document Based Design Implementation

5.1 Limitations of Relational Models

The relational database model maintains data integrity and structured querying capabilities; however it can introduce issues in scenarios requiring high flexibility and scalability [1].

Relational databases struggle with handling unstructured or semi-structured data [3], which is becoming more common especially in the agricultural context where satellite images and sensor data may be involved. The rigid nature of relational schemas makes it difficult to adapt to evolving data requirements without significant schema restructuring [1].

The limitations described emphasise the need to explore alternative database designs such as NoSQL document-based models, which are built for greater flexibility and scalability [3].

5.2 Alternative NoSQL Design

To overcome the constraints of the relational model, a document-based NoSQL database design could be introduced. NoSQL approaches focus on flexibility and scalability [3], making it well suited for handling unstructured and semi-structured data. Within this database, collections would contain key entities relating to farms, crops, resources, and initiatives, stored in JSON or BSON format, supporting nested structures and arrays.

Instead of relying on joins between tables through relationships, documents would embed related data within them. For example, a farm document could contain an array of crop documents, each with its own set of resources and initiatives. The denormalised approach reduces the need for complex joins, improving read performance and simplifying data retrieval [3], but can have some limitations presented in Section 5.3. Example collections are shown in Figure 2.

```
Farm data is stored within a farm collection as:
{
  "farmID": "1",
  "farm_location": "South Farm, Kent",
  "crops": [
    {
      "cropID": "101",
      "crop_name": "Wheat",
      "resources": [{ "resourceID": "1" }],
      "initiatives": [{ "initiativeID": "1" }]
    }
  ]
}
Initiative data is stored within an initiative collection as:
{
  "initiativeID": "1",
  "description": "Organic Farming"
}
```

Figure 2: Example NoSQL Document Structure for Farm and Initiative Collections

5.3 Comparative Analysis

While the NoSQL document-based design provides improvements in flexibility and scalability, it can introduce challenges compared to the relational approach. The relational model relies on vertical scaling, by enhancing the capacity of a single server, which could be more straightforward for smaller datasets and simpler applications. In contrast, NoSQL databases are horizontally scalable, with the ability to distribute data across multiple servers [3]. The NoSQL implementation enables nearly constant growth, making it suitable for deployment in the event the scale of the agricultural system expands.

However, the denormalised structure of NoSQL databases can lead to data redundancy and potential inconsistencies, as related data can be duplicated across multiple documents [3]. Duplication can complicate data management, especially when updates are required. For example, if a farm changed name or an initiative altered its description, that data will need to be altered across multiple documents. In contrast, within relational databases these changes can occur in a single location, the Farm or Initiative table respectively, improving consistency [1].

Schema flexibility presents another important consideration between the two approaches. Document-based databases can have dynamic schemas, meaning that different documents can have different data types and structures [3]. Conversely, relational databases require a fixed schema, which provides structure and consistency but can be less adaptable to alterations [1], which may be important if data requirements shift e.g. a farm needed to store unstructured data or additional attributes.

Performance also needs to be considered when weighing up the two approaches. Relational databases can excel in complex queries and transactions due to their structured nature and support for SQL [1]. NoSQL databases, on the other hand, provide superior performance for read-heavy workloads and large-scale data retrievals, as they are designed to handle high volumes of unstructured data efficiently [3].

The choice between approaches can be dependent on the data and project requirements. For the current agricultural system, the relational model is more suitable because the data has stable, well-defined structures. The system's primary use case involves complex analytical queries across multiple entities, which relational databases handle efficiently. However, if the system were to migrate towards handling high-volume, potentially unstructured data e.g. sensors or images, transitioning to a NoSQL approach may become advantageous.

References

[1] T. Connolly and C. Begg, *Database systems: A practical approach to design, implementation, and management*, 6th ed. Upper Saddle River, NJ, USA: Pearson, 2014.

[2] S. Selvaraj, "Introduction to restful apis," in *Mastering REST APIs*. United States: Apress L. P, 2024, pp. 1–18.

[3] A. Meier and M. Kaufmann, *SQL & NoSQL databases : models, languages, consistency options and architectures for big data management*. Wiesbaden: Springer Fachmedien Wiesbaden, 2019.