Goddit – reddit in Go

Name: Buster Antvorskov (buan0001)

Delivery date: 07/10/2025

To be added in final report:
- ➢ Group number

Table of Contents

1. Introduction

This report describes the “Goddit” project, a Reddit clone made in golang. “Clone” is here used quite liberally, as there’s many features not yet supported, along with a few features working differently to *the real thing*. At the end of the day Goddit isn’t much more than a forum with a few features added on top, so an interesting database design will mainly arise not through complexity granted by a multitude of tables, but by working with the data in interesting ways.

1.1 Problem description (+ architecture schema of the whole system)

System Schema is not ready yet. Will be filled out once a backend is attached.

1.2. Explanation of choices for databases and programming languages, and other tools.

No real technology choices have been made yet. Will be filled out once a backend is attached

2. Relational database

2.1. Intro to relational databases

Despite being one of the oldest technologies still used in modern software development, SQL databases are still the clear favourites when it comes to storing data in a structured manner. Although SQL databases have made great advances in the time since their first implementations, many of the principles have remained the same throughout the last 50 years.

The flexibility, speed and data integrity that normalization, along with the associated query language, provides have proven to be hard to beat.

In this section I wish to delve further into the details of relational databases, to examine their strengths and how they solve various relevant problems.

2.2. Database design

Will elaborate further in final report.

2.2.1. Entity/Relationship Model (Conceptual -> Logical -> Physical model)

See /docs in the project folder where both a conceptual and physical ER diagram are provided.

I will discuss them further in the final report

2.2.2. Normalization process

In order for a SQL database to be functional, the principle of normalization must be fulfilled. This isn't strictly a single principle, as there's multiple *levels* of normalization. However, they all share the same purpose of fracturing data into components, in such a way that no single field contains more than a single piece of information and in such a way that the same data is never duplicated in multiple places. There are multiple advantages of normalization, but namely:

- It guarantees data consistency by only having a "single source of truth". If a value that's referenced by a lot of entries changes, only one change is required as long as this value isn't nested inside each entry, but rather referenced by foreign key (FK).
- It makes it possible to combine different pieces of information in nearly endless ways. By separating the data into as atomic parts as possible, it's also possible, through the query language, to combine the data again in countless ways. This is highly advantageous when you rarely can foresee every useful way to represent the data upfront, as almost any future idea can be accommodated by a custom query.
- Related to the point of data consistency: It makes the database much easier to maintain, as a single change only ever has to be made in one place, not only easing the hassle of data changes for the developer, but also increasing performance, as fewer writes and searches through the database for the relevant information have to be performed.

2.3. Physical data model

2.3.1. Data types

The choice of data types in a database has significant implications for performance, storage, and data integrity. For example:

•**`INT`**: Used for numeric values such as IDs and counters. It is efficient for indexing and arithmetic operations but may waste space if smaller ranges (e.g., `TINYINT` or `SMALLINT`) suffice.

•**`VARCHAR`**: Allows variable-length strings, saving space compared to fixed-length types like `CHAR`. However, it may introduce slight overhead due to dynamic storage management.

•**`BOOLEAN`**: Represents true/false values but is often stored as `TINYINT` (1 byte) in MySQL, which can lead to confusion.

•**`ENUM`**: Useful for predefined sets of values (e.g., user roles). It is compact but less flexible if new values need to be added later.

•**`FLOAT`**: Used for monetary values (`amount_paid`). While it allows fractional values, it may introduce rounding errors. For precise calculations, `DECIMAL` is often preferred.

Size of the Types

•`INT`: 4 bytes, supports values from -2,147,483,648 to 2,147,483,647.

•`VARCHAR(n)`: Variable size, up to `n` characters + 1 or 2 bytes for length metadata.

•`BOOLEAN`: Stored as `TINYINT` (1 byte).

•`FLOAT(7,2)`: 4 bytes, with 7 digits of precision and 2 after the decimal point.

Taking into consideration the storage required for the data types can be quite relevant when the amount of entries grows into the millions. Putting a lower limit on the size of a VARCHAR field can also prove a valuable way to save on storage in that case.

2.3.2. Primary and foreign

- **Primary Keys**: Ensure each row in a table is uniquely identifiable. For example, `id` in `Users` and `Sub_goddits` serves as the unique identifier.
- **Foreign Keys**: Establish relationships between tables, enforcing referential integrity. For instance, `user_id` in `User_activity` references `Users(id)`.

How They Are Being Used

Primary Keys: Auto-incremented integers are used for most tables, ensuring unique and sequential identifiers.

Foreign Keys: Used extensively to model relationships, such as:

`sub_id` in `User_flairs` references `Sub_goddits(id)`.

`user_id` in `Subscriptions` references `Users(id)`.

These keys ensure that related data remains consistent and prevent orphaned records (e.g., a `User_activity` entry without a corresponding `Users` entry).

2.3.3. Indexes

Indexes improve query performance by allowing the database to locate rows more quickly. For example:

`INDEX idx_user_activity_user_id (user_id)` in `User_activity` speeds up queries filtering by `user_id`.

Speed Difference
Without indexes, the database performs a full table scan, which is inefficient for large datasets. With indexes, the database uses a B-tree or hash structure to locate rows in logarithmic time.

Trade-off in Storage
Indexes consume additional storage space and increase write overhead, since creating an index on a field essentially means the data will be duplicated. An additional downside is that updates and inserts may take longer due to index maintenance.

2.3.4. Constraints and referential integrity

Constraints like `FOREIGNKEY` and `CHECK` enforce normalization by ensuring data consistency. For example:

- `FOREIGN KEY(sub_id) REFERENCES Sub_goddits(id)` ensures that `User_flairs` entries are tied to valid `Sub_goddits`.

"Single Source of Truth"
Normalization ensures that data is not duplicated across tables, reducing redundancy. For example:

- The `Subscriptions` table links `Users` and `Sub_goddits`, avoiding the need to store subscription data in both tables.

No “Dangling” Data

Referential integrity prevents "dangling" data. For instance:

- `ON DELETE CASCADE` ensures that when a `Sub_goddits` entry is deleted, related `User_flairs` and `Subscriptions` entries are also removed.

2.4. Stored objects – stored procedures / functions, views, triggers, events

Database programming involves embedding logic within the database itself, while application programming handles logic in the application layer. For example:

- **Database Programming**: The `update_daily_visitors` event recalculates `daily_users` directly in the database.
- **Application Programming**: The same logic could be implemented in the application, requiring additional queries.

Pros:

- **Reduced Network Traffic**: Operations like `update_daily_visitors` are executed entirely within the database, reducing the need for multiple queries from the application
- **Centralized Logic**: Triggers like `after_subscribe` ensure consistent behavior across all applications interacting with the database.

Cons:

- **Harder to Debug**: Errors in stored objects like triggers or events can be harder to trace compared to application code.
- **Limited Flexibility**: Changes to stored objects require database-level access and may disrupt live systems.

2.5. Transactions. Explanation of the structure and implementation of transactions

Not yet implemented

2.6. Auditing. Explanation of the audit structure implemented with triggers

Not yet implemented

2.7. Security.

Not yet implemented

2.7.1. Explanation of users and privileges

Not yet implemented

2.7.2. SQL Injection – what is it and how it is dealt with in the project?

Not yet implemented

2.8. Description of the CRUD application for RDBMS

Not yet implemented