1. Intro & Design Twitter

Twitter Clone - Step-by-Step ERD & Database Design Notes

This document explains the **entity-relationship diagram (ERD)** design for a **Twitter-like social media app**. The goal is to build a clear understanding of database modeling by starting from **feature requirements**, then building **intuitively** and **iteratively** step-by-step.

🎯 Purpose of this Document

- Designed for **SDE-1 interview prep** and **self-learning**
- Follows an **intuition-first** approach before implementation
- Breaks down complex concepts into digestible steps
- SQL/PostgreSQL-centric thinking (constraints, structure, integrity)

✅ Step 1: Requirements Breakdown

Let's begin by **understanding the application's core features**:

🔍 Feature Set:

- Users can **sign up**
- Users can **post tweets** (text + optional media)
- Users can **follow other users**
- Users can **like tweets**
- Users can **comment** on tweets
- Users can **subscribe to premium (e.g., blue checkmark)**

From this, we extract the **main entities** and **actions**:

📄 Core Entities (Tables):

1. Users
2. Tweets
3. Media (attached to tweets)
4. Follows (many-to-many)
5. Likes
6. Comments
7. Subscriptions

🧠 Step 2: Understanding Primary Keys & Business Logic

🔑 What is a Primary Key?

A **primary key (PK)** is a column (or set of columns) that **uniquely identifies each row** in a table.

Properties of a Primary Key:

- Must be **unique** across the table
- Cannot be **NULL**
- Should be **immutable** (doesn't change over time)
- Optimized for fast lookups (often indexed by default)

⚠️ What is Business Logic in a Database?

Business logic refers to **real-world rules and decisions** related to your application's features or workflows.

Examples in our app:

- `username` is chosen by users
- `email` is required for login/communication
- `plan_type` decides user privileges

🚫 Why Not Use Business Logic as a Primary Key?

1. **User-controlled and changeable**
   - A user may want to change their `username` or `email`
   - Changing PKs causes complex cascading updates
2. **Validation-heavy**
   - Requires frequent uniqueness checks at the app level
   - Email format can vary or cause issues (e.g., uppercase vs lowercase)
3. **Security implications**
   - Exposing sensitive or meaningful data (like email) via URLs or APIs is dangerous
4. **Lack of global uniqueness**
   - Business values often assume uniqueness *per context* (e.g., `username` in one org)

✅ Preferred Approach: Use a Surrogate Key

- Use a **UUID** (universally unique identifier) or **CUID** as `id`
- Auto-generated by backend or database

Example in PostgreSQL:

```
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    username VARCHAR(255) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    bio TEXT,
    created_at TIMESTAMP DEFAULT NOW()
);
```

Here, `id` is the primary key — not `username` or `email`.

🔁 Step 3: Start with the Most Fundamental Entity - `Users`

Every action in our system starts with a user — posting tweets, liking, following, etc.

Key Points:

- `id` is the primary key (UUID)
- `username` and `email` should be unique
- Always include `created_at`

We'll keep the design simple and scalable — more fields can be added later based on use case.

🧳 Step 4: Moving Forward - Planning Relationships

Before jumping into the rest of the tables, let's build strong **intuition** for:

- **One-to-One**: e.g., A user has one subscription
- **One-to-Many**: e.g., A user can post many tweets
- **Many-to-Many**: e.g., Users follow each other

We'll explore these as we move entity-by-entity. No code for ERD generation yet — we'll focus on logic first.

Stay tuned for:

- Tweets
- Media
- Comments
- Likes
- Follows
- Subscriptions

Each will be broken down step-by-step with **justification** before implementing any code.

📌 What's Next

Next, we will:

- Go entity by entity
- Understand **real-world behavior** first
- Justify each relationship (1:1, 1:N, N:N)
- Plan for constraints, normalization, scalability

🦫 Step 5: Tweets Entity - Thinking in Relationships

💡 Real-World Behavior

- A user can post **many tweets**
- A tweet can only be created by **one user**

This is a textbook **One-to-Many (1:N)** relationship:

- One `user` → many `tweets`
- Each `tweet` → one `user`

🛠️ Database Design Thoughts:

- `tweets` table should have a foreign key `user_id` referring to `users(id)`
- This allows us to query: *"show me all tweets by this user"*
- Every tweet should be timestamped (`created_at`)
- Optional: add `updated_at` if we allow tweet edits

⚙️ Suggested Fields in `tweets`:

Column	Type	Constraints
id	UUID	PRIMARY KEY
user_id	UUID	FOREIGN KEY → users(id)
content	TEXT	NOT NULL
created_at	TIMESTAMP	DEFAULT NOW()

Column	Type	Constraints
updated_at	TIMESTAMP	NULLABLE (if edits allowed)

Example Query:

```
CREATE TABLE tweets (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID NOT NULL REFERENCES users(id),
    content TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP
);
```

🖼️ Step 6: Media Entity - Handling Tweet Attachments

💡 Real-World Behavior

- A tweet can contain **zero, one, or multiple media files** (images, videos, gifs)
- Each media item **belongs to exactly one tweet**

This leads to a **One-to-Many (1:N)** relationship:

- One `tweet` → many `media`
- Each `media` → one `tweet`

🛠️ Database Design Thoughts:

- `media` table stores URL and type of media (image, video, etc.)
- Foreign key `tweet_id` refers to the `tweets(id)`
- Include timestamps for tracking uploads
- Optional: a `position` field if media has ordering (e.g., first image, second image)

⚙️ Suggested Fields in `media`:

Column	Type	Constraints
id	UUID	PRIMARY KEY
tweet_id	UUID	FOREIGN KEY → tweets(id)
url	TEXT	NOT NULL
media_type	TEXT	e.g., 'image', 'video', 'gif'
created_at	TIMESTAMP	DEFAULT NOW()

Example Query:

```
CREATE TABLE media (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    tweet_id UUID NOT NULL REFERENCES tweets(id),
    url TEXT NOT NULL,
    media_type TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT NOW()
);
```

In the next section, we'll move to the **comments** entity and understand the dual relationship it has with both users and tweets.

🗨️ Step 7: Comments Entity - Dual Relationships

In any social media platform, **comments** are crucial for user interaction and engagement.

We'll now explore how to model the `comments` entity logically and relationally in our database.

💡 Real-World Behavior

- A **user** can post **many comments**
- A **tweet** can receive **many comments**
- A **comment** must be associated with **exactly one user** (who authored it)
- A **comment** must also belong to **exactly one tweet** (the thread it's under)

💬 Two Distinct Relationships

This makes `comments` a **junction entity** that connects:

1. `users` → `comments` (1:N)
2. `tweets` → `comments` (1:N)

➕ Optional Add-ons

- We might want to allow nested replies (comments on comments), but we'll leave that out for now for simplicity.

🛠️ Database Design Thoughts

The `comments` table should:

- Have a unique ID (UUID)
- Store comment content (`TEXT`)
- Store `user_id` as a foreign key to reference the author
- Store `tweet_id` as a foreign key to reference the tweet
- Include a `created_at` timestamp

⚙️ Suggested Fields in `comments`:

Column	Type	Constraints
id	UUID	PRIMARY KEY
user_id	UUID	FOREIGN KEY → users(id)
tweet_id	UUID	FOREIGN KEY → tweets(id)
content	TEXT	NOT NULL
created_at	TIMESTAMP	DEFAULT NOW()

🧠 Why This Design Makes Sense

- Each `comment` must be traced back to both:
  - Who created it? → `user_id`

    - On which tweet? → `tweet_id`
- This setup allows queries like:
    - "Get all comments on this tweet"
    - "Get all comments made by this user"
- Ensures **data normalization** and **referential integrity**

SQL Snippet (for reference)

```
CREATE TABLE comments (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID NOT NULL REFERENCES users(id),
    tweet_id UUID NOT NULL REFERENCES tweets(id),
    content TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT NOW()
);
```

In the next section, we'll tackle the **likes** entity, which connects users and tweets in a many-to-many relationship.

We'll explore how to structure a clean, efficient join table for it.

❤️ Step 8: Likes Entity - Many-to-Many Relationships

Likes are a fundamental part of user interaction in any social media app. In a Twitter-like system, users can like multiple tweets, and each tweet can be liked by multiple users.

This is a **classic many-to-many (M:N) relationship** between `users` and `tweets`.

💡 Real-World Behavior

- A **user** can like **many tweets**
- A **tweet** can be liked by **many users**
- Each **like** is specific to one `user` and one `tweet`

Thus, we need a **join table** to map this relationship.

🔗 Why a Separate Likes Table?

Many-to-many relationships should not be modeled by directly embedding arrays in relational DBs.

Instead, we use a join table — in this case, `likes`.

Each row in the `likes` table represents **one unique like action** performed by a user on a tweet.

🛠️ Database Design Thoughts

The `likes` table should:

- Have its own `id` as primary key (UUID)
- Store `user_id` (who liked)
- Store `tweet_id` (which tweet was liked)
- Have a `created_at` timestamp

> Optionally, you can enforce a unique constraint on (user_id, tweet_id) to prevent duplicate likes.

⚙️ Suggested Fields in `likes`:

Column	Type	Constraints
id	UUID	PRIMARY KEY
user_id	UUID	FOREIGN KEY → users(id)
tweet_id	UUID	FOREIGN KEY → tweets(id)
created_at	TIMESTAMP	DEFAULT NOW()

🧠 Why This Design Makes Sense

- Prevents duplication of likes with a `(user_id, tweet_id)` uniqueness constraint
- Cleanly tracks who liked what and when
- Supports queries like:
  - "Get all tweets liked by a user"
  - "Count how many likes a tweet received"
  - "Check if a specific user liked a tweet"

SQL Snippet (for reference)

```
CREATE TABLE likes (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID NOT NULL REFERENCES users(id),
    tweet_id UUID NOT NULL REFERENCES tweets(id),
    created_at TIMESTAMP DEFAULT NOW(),
    CONSTRAINT unique_like UNIQUE (user_id, tweet_id)
);
```

In the next section, we'll handle the **follows** relationship — a more complex **self-referencing many-to-many** relationship between users.

🔁 Step 9: Follows Entity - Self-Referencing Many-to-Many

Following relationships are at the heart of a social network like Twitter. This feature allows users to follow other users, resulting in a **many-to-many** relationship **between users themselves**.

💡 Real-World Behavior

- A **user** can follow **many other users**
- A **user** can be followed by **many users**

This clearly represents a **self-referencing many-to-many relationship**.

🧠 How to Represent This in SQL

We can't use a direct relationship within the `users` table for this. Instead, we create a **join table**, typically named `follows`.

This table records each follow action:

- Who is the **follower**?
- Who is the **followee**?

Each row represents: *User A follows User B*

🛠️ Database Design Thoughts

The `follows` table should:

- Have a unique `id` (UUID)
- Have two foreign keys:
  - `follower_id` → users(id)
  - `followee_id` → users(id)
- Include a `created_at` timestamp
- Enforce a uniqueness constraint on `(follower_id, followee_id)`

⚙️ Suggested Fields in `follows`:

Column	Type	Constraints
id	UUID	PRIMARY KEY
follower_id	UUID	FOREIGN KEY → users(id)
followee_id	UUID	FOREIGN KEY → users(id)
created_at	TIMESTAMP	DEFAULT NOW()

🔐 Constraints to Consider

- Prevent duplicate follow records with a unique pair constraint

- Prevent users from following themselves

Additional Constraints (Recommended)

```
CHECK (follower_id <> followee_id)
```

🧠 Why This Design Makes Sense

- Clearly models "who follows whom"
- Efficient for querying both followers and followees
- Prevents redundant or self-follow entries

Supports queries like:

- “List all followers of a user”
- “List everyone this user is following”
- “Count followers/following”

SQL Snippet (for reference)

```
CREATE TABLE follows (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    follower_id UUID NOT NULL REFERENCES users(id),
    followee_id UUID NOT NULL REFERENCES users(id),
    created_at TIMESTAMP DEFAULT NOW(),
    CONSTRAINT unique_follow UNIQUE (follower_id, followee_id),
    CONSTRAINT no_self_follow CHECK (follower_id <> followee_id)
);
```

Next up, we’ll finalize with the **subscriptions** entity — representing the optional premium plan for users.

💎 Step 10: Subscriptions Entity - One-to-One Relationship

In modern social media platforms, offering premium features (like a verified badge) is common. In our Twitter-like system, this is handled by **subscriptions**.

Each user can have **at most one active subscription**, making this a **one-to-one relationship** between `users` and `subscriptions`.

💡 Real-World Behavior

- A **user** can optionally subscribe to a premium plan
- Each **subscription** belongs to exactly **one user**
- A **user** can have **only one active subscription** at a time

This is modeled as:

- One `user` → one `subscription`
- One `subscription` → one `user`

🛠️ Database Design Thoughts

The `subscriptions` table should:

- Have a unique `id` (UUID)
- Include a `user_id` field (foreign key and unique)
- Store plan details (e.g., plan type)
- Include start and end timestamps
- Optional: `is_active` boolean flag

⚙️ Suggested Fields in `subscriptions`:

Column	Type	Constraints
id	UUID	PRIMARY KEY
user_id	UUID	FOREIGN KEY → users(id), UNIQUE
plan_type	TEXT	e.g., 'basic', 'premium', 'verified'
start_date	DATE	NOT NULL

Column	Type	Constraints
end_date	DATE	NULLABLE (null means active)
created_at	TIMESTAMP	DEFAULT NOW()

🧠 Why This Design Makes Sense

- Enforces **one-to-one** via a `UNIQUE` constraint on `user_id`
- Can easily check if a user is subscribed
- Allows future enhancements:
  - Plan history table
  - Grace periods, renewals
  - Payment linkage

SQL Snippet (for reference)

```
CREATE TABLE subscriptions (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID UNIQUE NOT NULL REFERENCES users(id),
    plan_type TEXT NOT NULL,
    start_date DATE NOT NULL,
    end_date DATE,
    created_at TIMESTAMP DEFAULT NOW()
);
```

With this, we've completed the high-level database modeling of all core entities in our Twitter-like application. Ready to proceed with indexing, constraints, ERD generation, or additional features!

💎 Step 10: Subscriptions Entity - One-to-One Relationship

In modern social media platforms, offering premium features (like a verified badge) is common. In our Twitter-like system, this is handled by **subscriptions**.

Each user can have **at most one active subscription**, making this a **one-to-one relationship** between `users` and `subscriptions`.

💡 Real-World Behavior

- A **user** can optionally subscribe to a premium plan
- Each **subscription** belongs to exactly **one user**
- A **user** can have **only one active subscription** at a time

This is modeled as:

- One `user` → one `subscription`
- One `subscription` → one `user`

🛠️ Database Design Thoughts

The `subscriptions` table should:

- Have a unique `id` (UUID)
- Include a `user_id` field (foreign key and unique)
- Store plan details (e.g., plan type)
- Include start and end timestamps
- Optional: `is_active` boolean flag

⚙️ Suggested Fields in `subscriptions`:

Column	Type	Constraints
id	UUID	PRIMARY KEY
user_id	UUID	FOREIGN KEY → users(id), UNIQUE
plan_type	TEXT	e.g., 'basic', 'premium', 'verified'
start_date	DATE	NOT NULL

Column	Type	Constraints
end_date	DATE	NULLABLE (null means active)
created_at	TIMESTAMP	DEFAULT NOW()

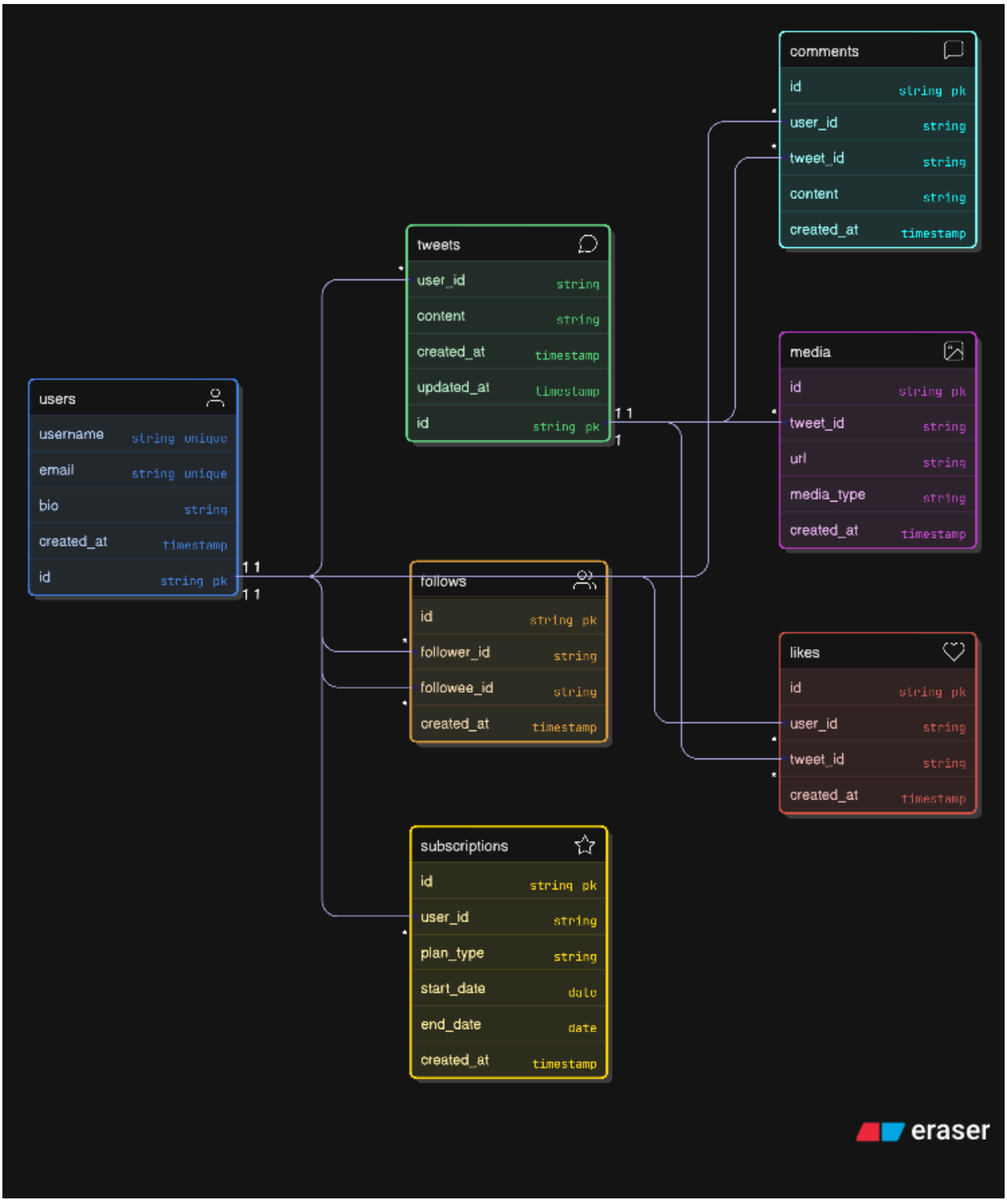
🧠 Why This Design Makes Sense

- Enforces **one-to-one** via a `UNIQUE` constraint on `user_id`
- Can easily check if a user is subscribed
- Allows future enhancements:
  - Plan history table
  - Grace periods, renewals
  - Payment linkage

SQL Snippet (for reference)

```
CREATE TABLE subscriptions (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID UNIQUE NOT NULL REFERENCES users(id),
    plan_type TEXT NOT NULL,
    start_date DATE NOT NULL,
    end_date DATE,
    created_at TIMESTAMP DEFAULT NOW()
);
```

With this, we've completed the high-level database modeling of all core entities in our Twitter-like application. Ready to proceed with indexing, constraints, ERD generation, or additional features!



🧠 Deep-Dive Tasks for the Reader

🔷 Task 1: Add `retweets` Feature

> In Twitter, users can retweet other users' tweets.

Design a `retweets` table that:

- Tracks `user_id` (who retweeted)
- Tracks `original_tweet_id` (which tweet was retweeted)
- Includes `created_at` timestamp

Challenge: Can a retweet also have a comment (like quote-retweet)? Model that case.

🔷 Task 2: Normalize Media Types

> Instead of storing media types as free text, move them to a new table.

Steps:

- Create a `media_types` table (`id`, `name`)
- Link `media` to `media_types` via foreign key

This is useful for enums or controlled vocabularies.

🔷 Task 3: Add Indexing Strategy

> Interviewers love asking: “How would you improve performance?”

Try this:

- Identify which fields should be indexed.
- Think about queries: search by `username`, `tweet_id`, etc.
- Add indexes in SQL or write index plans.

🔷 Task 4: Create an Analytics Table

> For example: tracking how many tweets a user makes per day.

Your challenge:

- Design a `user_activity` table that stores:
  - `user_id`
  - `date`
  - `tweet_count`
  - `like_count`
- Think: Should this be precomputed or live queried?

🔷 Task 7: Design Query Challenges

> Imagine you're the backend dev. Write raw SQL for the following:

1. Get all tweets by a user with their media.
2. Count how many likes a tweet has.
3. Find all users who follow each other mutually.
4. List users with active subscriptions.
5. Get the top 5 most commented tweets in the last week.

🔷 Task 8: Extend for Notifications

> Users get notified when someone likes, comments, or follows them.

Design a `notifications` table:

- Fields: `id`, `recipient_user_id`, `type`, `actor_user_id`, `related_tweet_id`, `created_at`, `read`
