Udiddit: Data Model Report

Udiddit (**UDT**) is a social news aggregation website. In addition to this, UDT also provides a content rating system to its users, and UDT serves as a discussion place for its users, covering a vast array of topics. In order to make use of UDT's services, a person must first register as a user on UDT. Once a person is registered as a user, they are able to create posts on UDT, containing either original textual content or external links to existing content found elsewhere on the internet.

Users also have the ability to comment on existing posts created by other users, and engage in discussions regarding these posts in this way. Finally, UDT users can either upvote (or 'like') or downvote (or 'dislike') posts that are submitted/created by other users. Due to the sheer amount of user interactivity and user engagement, UDT will presumably generate and store large amounts of data.

Due to the tight deadlines and schedules involved in the development of UDT, the Postgres (**PG**) data model being used by UDT at this moment in time has not been designed efficiently. Consequently, it has the potential to create issues in the future. This reports aims to first identify any/all potential issues within the existing PG data model, and then create a new data model that will remedy any issues that were identified while also being efficient at the same time.

1. Existing Data Model Investigation

At the outset, it must be noted that when using a Relational Database (**RDB**) such as PG, there is a certain preferred format which should be used when storing data, in order to allow the RDB (such as PG) to work on the data quickly and efficiently. This is achieved through a process referred to as Data Normalisation (**DNMS**).

The **first issue** that has been identified in the data model, being the "bad_comments" (**BC**) and the "bad_posts" (**BP**) tables, is the existence of **multiple entities** within both the BC and BP tables. Based on my analysis, there are currently 3 distinct entities within the BC table.[1] Within the BP table, there are 4 distinct

[1] The entities are **comments**, **users** and **posts**.

entities.[2] The presence of multiple entities within in these 2 tables could potentially result in insertion, update and deletion anomalies occurring as a result of any modification that is made to either of the tables.[3]

I would recommend the following to resolve this issue:

- Create a separate table for each entity identified above.[4]
- Within each table, implement an *id* column to be used as the Primary Key for each value stored in the table.

By implementing the above recommendations, should a user change their username or delete their account, such a modification would not have to be updated in any of the other tables except the **users** table. This will also reduce the duplication of data within the data model itself and will also decrease the likelihood of any human error occurring when updating the data model to reflect any modifications.

The **second issue** that has been picked up relates to the **users** of UDT. As it stands, there is no database within the data model which contains any information in relation to users. Currently, there is no way for UDT to easily track the number of users on their platform, and there is no information provided for these users other than their usernames.

To resolve this issue, I would recommend the following:

- Implement the **first recommendation** mentioned above in order to have a separate user table.
- Create columns within this user table which will hold data such as a user's *username*, *email address*, *date of birth* and any other relevant information.
- Implement the necessary data constraints to ensure that data such as *username* and *email address* are not duplicated.

The abovementioned recommendation will provide UDT with a database of their users, which will be crucial for analytical and statistical purposes going forward with the platform. Also, through the use of the correct data constraints, this recommendation will also prevent multiple users having the same username or using the same email address to register with on UDT.

The **third issue** that has been identified pertains to the *text_content* column which is appears in both tables. The problem with this column is that there is **no maximum**

---

2 The entities are **posts**, **topics**, **users**, and **votes**.

3 For example, if a user decides to change their username or delete their account, this modification will have to be reflected in both tables, as the **username** (as an entity or data object) appears in both tables. This would result in time and effort being expended in order to update the tables to reflect this change.

4 The entities that should each have a separate table are **users**, **posts**, **topics**, **comments**, and **votes**.

character length that can be imposed on the data type for this column.[5] Without a reasonable character limit for the content of both posts and comments, this could result in the intentional or unintentional abuse of these functions by UDT users. Additionally, data storage capacity could be stretched or become insufficient where there is no character limit for textual data.

This issue can be resolved in the following way:

- Impose a character limit post of 40 000 characters for the *text_content* column in the posts table.
- Impose a character limit post of 10 000 characters for the *text_content* column in the comments table.

The **fourth issue** that is present within the data model is the current way that the **voting system** on **posts** has been implemented. At present, the upvotes and downvotes columns contain more than 1 value within each of their respective cells, with each cell containing a list of users who have either upvoted or downvoted a specific post.

This issue can be fixed in the following manner:

- Implement the **first recommendation** outlined above in order to create a separate table for upvotes and downvotes.
- Create *post_id* and *user_id* columns in the table.
- For each unique *post_id* value, assign a separate value for each user that upvoted or downvoted that post in the table.

Through the implementation of the above recommendation, UDT will find it easier to calculate the number of users that have upvoted and downvoted a specific post, as the upvotes and downvotes will be contained in a separate table. This will also assist in the normalisation of the original data by restricting the cells within the tables to having only one value.

The **fifth issue** that I have found within the data model concerns the **character limit** for the *url* column within the BP table. At present, the character limit for this column is 4 000 characters. While this is technically sound, I would recommend a character limit of 2 048 characters for URLs[6] as this seems to be the standard that is commonly used on the internet across most web browsers.

Finally, the **sixth issue** that appears from the data model is data **inconsistency**. Within the BP table, the *id* column is assigned the **SERIAL** data type, which is a medium integer.[7] In the BC table, the *post_id* column is assigned the **BIGINT** data

[5] The data type in question for both tables being **TEXT**.
[6] Uniform Resource Locator.
[7] Which ranges from -2 147 483 648 to 2 147 483 647.

type, which is a large integer.[8] This is a clear case of a date referential inconsistency, as the *post_id* column refers to values found in the *id* column.

Additionally, there seems to be another **inconsistency** regarding the values contained in the *topic* column in the BP table. There are values which contain multiple words which are separated either using underscores or hyphens.

To eliminate these inconsistencies, I would recommend the following:

- UDT must decide on the range of the **SERIAL** value assigned to the *id* column within the BP table.
- After selecting the correct **SERIAL** value, the *post_id* column data type must be changed to be consistent with the chosen **SERIAL** value.[9]
- UDT must decide on the rules that apply to the *topic* column values which contain multiple words, specifically rules that indicate the character to be used in separating these multiple word values.

By implementing these recommendations outlined in this section of the report, UDT can take a step closer to DNMS to ensure that their data model can operate quickly and efficiently.

2. New Udiddit Schema

To implement the all recommendations outlined above, UDT will need to create a completely new data schema. The existing UDT data schema is not fit to handle all the modifications that need to be made without considerable effort being dedicated to modifying the existing table structures and updating the existing data. It would be an easier task to simply create a new data schema and use that as the platform going forward.

As per the guidelines UDT has issued, the following is the Database Definition Language (**DDL**) code for the tables that make up the new UDT schema:[10]

```
CREATE TABLE "users" (
  "id" SERIAL,
  "username" VARCHAR(25) NOT NULL,
  CONSTRAINT "users_pk" PRIMARY KEY ("id"),
  CONSTRAINT "unique_usernames" UNIQUE ("username"),
```

[8] Which ranges from -9 223 372 036 854 775 808 to 9 223 372 036 854 775 807.

[9] For example, as it currently stands, the correct data type that should be used for the *post_id* column is **INTEGER**. **BIGINT** can only be used in conjunction with the **BIGSERIAL** data type.

[10] Please take note the tables must be created in the order that they are presented within this section.

```
  CONSTRAINT "no_empty_usernames"
  CHECK (LENGTH(TRIM("username")) > 0)
);
```

Table 1 DDL for the "users" table.

```
CREATE TABLE "topics" (
  "id" SERIAL,
  "name" VARCHAR(30) NOT NULL,
  "description" TEXT,
  CONSTRAINT "topics_pk" PRIMARY KEY ("id"),
  CONSTRAINT "unique_topic_names" UNIQUE ("name"),
  CONSTRAINT "no_empty_topic_names"
  CHECK (LENGTH(TRIM("name")) > 0),
  CONSTRAINT "description_limit"
  CHECK (LENGTH("description") <= 500)
);
```

Table 2 DDL for the "topics" table.

```
CREATE TABLE "posts" (
  "id" SERIAL,
  "topic_id" INTEGER,
  "user_id" INTEGER,
  "title" VARCHAR(100),
  "url" VARCHAR(2048),
  "text_content" TEXT,
  CONSTRAINT "posts_pk" PRIMARY KEY ("id"),
  CONSTRAINT "topics_id_fk" FOREIGN KEY ("topic_id")
  REFERENCES "topics" ON DELETE CASCADE,
  CONSTRAINT "users_id_fk" FOREIGN KEY ("user_id")
  REFERENCES "users" ON DELETE SET NULL,
  CONSTRAINT "no_empty_titles"
  CHECK (LENGTH(TRIM("title")) > 0),
  CONSTRAINT "post_content_restriction"
  CHECK (("url" IS NOT NULL AND "text_content" IS NULL) OR
        ("url" IS NULL AND "text_content" IS NOT NULL)),
  CHECK (LENGTH("text_content") <= 40000)
);
```

Table 3 DDL for the "posts" table.

```
CREATE TABLE "comments" (
  "id" SERIAL,
  "user_id" INTEGER,
  "post_id" INTEGER,
  "comment_id" INTEGER,
  "text_content" TEXT,
  CONSTRAINT "comments_pk" PRIMARY KEY ("id"),
  CONSTRAINT "username_fk" FOREIGN KEY ("user_id")
  REFERENCES "users" ON DELETE SET NULL,
  CONSTRAINT "posts_fk" FOREIGN KEY ("post_id")
  REFERENCES "posts" ON DELETE CASCADE,
```

```
  CONSTRAINT "comment_thread_fk" FOREIGN KEY ("comment_id")
  REFERENCES "comments" ("id") ON DELETE CASCADE,
  CONSTRAINT "no_empty_comments"
  CHECK (LENGTH(TRIM("text_content")) > 0),
  CONSTRAINT "max_comment_character_length"
  CHECK (LENGTH("text_content") <= 10000),
  CONSTRAINT "no_double_comments"
  CHECK (("post_id" IS NULL AND "comment_id" IS NOT NULL) OR
        ("post_id" IS NOT NULL AND "comment_id" IS NULL))
);
```

Table 4 DDL for the "comments" table.

```
CREATE TABLE "votes" (
  "post_id" INTEGER REFERENCES "posts" ON DELETE CASCADE,
  "user_id" INTEGER REFERENCES "users" ON DELETE SET NULL,
  "vote" INTEGER NOT NULL,
  CONSTRAINT "user_voting_rule"
  UNIQUE ("user_id", "post_id"),
  CONSTRAINT "up_down_votes"
  CHECK ("vote" = 1 OR "vote" = -1)
);
```

Table 5 DDL for the "votes" table.

As requested by UDT, the following Database Query Language (**DQL**), DDL and Database Manipulation Language (**DML**) code will assist in achieving most of UDT's web analytic objectives set out in **Guideline 2**:

Guideline 2(a) – List all users who haven't logged in during the last year

```
ALTER TABLE "users"
  ADD COLUMN "login_date" TIMESTAMP;
```

Table 6 DDL for Guideline 2(a).

```
CREATE INDEX "login_date_index"
    ON "users" ("login_date");
```

Table 7 DDL for Guideline 2(a).

Guideline 2(b) – List all users who haven't created any posts

```
CREATE INDEX "user_id__index"
  ON "posts" ( "user_id");
```

Table 8 DDL for Guideline 2(b).

Guideline 2(d) – List all topics that don't have any posts

```
CREATE INDEX "topic_id_index"
  ON "posts" ("topic_id");
```

Table 9 DDL for Guideline 2(d).

Guideline 2(h) – Find all posts that link to a specific URL, for moderation purposes

```
CREATE INDEX "url_index"
  ON "posts" ("url");
```

Table 10 DDL for Guideline 2(h).

Guideline 2(i) – List all the top-level comments for a given post

```
CREATE INDEX "post_id_index"
  ON "comments" ("post_id");
```

Table 11 DDL for Guideline 2(i).

Guideline 2(k) – List the latest 20 comments made by a user

```
CREATE INDEX "user_id_index"
  ON "comments" ("user_id");
```

Table 12 DDL for Guideline 2(k).

Please ensure to create all the indexes listed above before commencing with the migration of UDT's existing data from the BC and BP table.

3. Existing Data Migration

Once the new UDT data schema is in place, the next task involves the migration of UDT's existing data, from the BC and BP tables, into the new tables. Again, the order in which the migration is performed is important. The migration of the existing data will be facilitated by the use of DML.

As per the guidelines issued by UDT, the following is all the DML code relating to the migration of UDT's existing data:

```
INSERT INTO "users" ("username")
  SELECT DISTINCT "username"
  FROM "bad_posts"

  UNION

  SELECT DISTINCT "username"
  FROM "bad_comments"

  UNION

  SELECT DISTINCT REGEXP_SPLIT_TO_TABLE("upvotes", ',') AS username
  FROM "bad_posts"

  UNION
```

```
  SELECT DISTINCT REGEXP_SPLIT_TO_TABLE("downvotes", ',') AS username
  FROM "bad_posts";
```

Table 13 DML for the migration of "users" data.

```
INSERT INTO "topics" ("name")
  SELECT DISTINCT "topic"
  FROM "bad_posts";
```

Table 14 DML for the migration of "topics" data.

```
INSERT INTO "posts" ("id", "topic_id", "user_id", "title", "url",
"text_content")
  SELECT "bp"."id",
         "t"."id",
         "u"."id",
         LEFT("bp"."title", 100),
         "bp"."url",
         "bp"."text_content"
         FROM "bad_posts" "bp"
         LEFT JOIN "topics" "t"
         ON "t"."name" = "bp"."topic"
         JOIN "users" "u"
         ON "u"."username" = "bp"."username";
```

Table 15 DML for the migration of "posts" data.

```
INSERT INTO "comments" ("user_id", "post_id", "text_content")
  SELECT "u"."id" AS user_id,
         "p". "id" AS post_id,
         "bc"."text_content"
         FROM "bad_comments" "bc"
         JOIN "users" "u"
  ON "u"."username" = "bc"."username"
  JOIN "posts" "p"
  ON "p"."id" = "bc"."post_id";
```

Table 16 DML for the migration of "comments" data.

```
INSERT INTO "votes" ("post_id", "user_id", "vote")
  SELECT sub1."id" AS post_id,
         "u"."id" AS user_id,
         1 AS vote
         FROM (
           SELECT "id",
           REGEXP_SPLIT_TO_TABLE("upvotes", ',') AS upvoters
           FROM "bad_posts"
         ) sub1
         JOIN "users" "u"
         ON "u"."username" = sub1."upvoters";
```

Table 17 DML for the migration of "upvotes" data.

```
INSERT INTO "votes" ("post_id", "user_id", "vote")
```

```
SELECT sub1."id" AS post_id,
       "u"."id" AS user_id,
       -1 AS vote
       FROM (
         SELECT "id",
         REGEXP_SPLIT_TO_TABLE("downvotes", ',') AS downvoters
         FROM "bad_posts"
       ) sub1
       JOIN "users" "u"
       ON "u"."username" = sub1."downvoters";
```

Table 18 DML for the migration of "downvotes" data.