Explanation and Model Solution for "Assignment 3: The Relational Data Model and SQL"

Jelle Hellings

3DB3: Databases – Fall 2021

Department of Computing and Software
McMaster University

Foreword

The model solution for part one consists of both a rational for our solution and the solution itself. The model solution for part two only consists of a solution, as this is sufficient to explain the approach.

Part one: The social media features

Rational:

As the first step, we analyze the description of the ER-diagram of Figure 1 to find any information relevant for our translation. In this analysis, we use the following text annotations:

an entity	The highlighted piece of text resembled an entity-like object.
an attribute	The highlighted piece of text resembled an attribute of some entity-like object.
A RELATIONSHIP	The highlighted piece of text resembled a relationship between entity-like objects.
`a constraint`	The highlighted piece of text resembled a constraint on data.
~~not-relevant~~	The highlighted piece of text is not relevant (e.g., context, application details, ...)

Following is the original description:

> ~~The social media features center around its~~ **users** (the subscribers).[1] Users have *usernames*, which are not necessary unique. `To distinguish users with the same username, the consultant proposed to assign each such user a different number`.[2] Users will log in using their `unique email address`[3] and a *password*. According to the consultant, the password should not be stored in plain text, `but as a pair of a 512-bit (64-byte) hash and a 64-byte salt value`.[4] ~~Together, the hash and salt are sufficient to determine whether a login attempt provided the right password, this without providing easy access to the stored password (even if the system gets compromised).~~
>
> The social media features break down in three types:

[1] The description uses the term "user" for **Subscriber** entities.
[2] Hence, the attributed *username* and *number* are the primary key.
[3] An additional constraint: the attribute *email* should be unique.
[4] Type information for *hash* and *salt*: they should be **BINARY**`(64)`, which is not supported by our version of DB2. Hence, we use the DB2-specific **CHAR**`(64)` **FOR BIT DATA**.

- USERS CAN *BEFRIEND* OTHER USERS.[5] ~~If a user X indicates that Y is a friend, then this indicates that X follows the activities of Y, but this does not imply that Y is also a friend of X.~~[6]
- Users can ***review*** **films**. Each review assigns a score to the film and reviews can be *revised* with more information.[7] Each review can optionally have a video component (e.g., as in a vlog) and optionally have a text component (e.g., as in a blog).[8]
- Users can place ***reactions*** on REVIEWS and on other REACTIONS (a reaction is either on a single review or on a single reaction, but not on both).[9]

~~The ER-Diagram for the social media feature can be found in Figure 1.~~

We will use the information collected during this analysis in our solution.

Solution:

We start with the *Subscriber* entity, as it is not a weak entity and does not partake in any one-to-many relationships. Translating this entity yields the following relational schema:

Subscriber(<u>username</u> : **VARCHAR**(100), <u>number</u> : **INT**, email : **VARCHAR**(200), hash : **BINARY(64)**, salt : **BINARY(64)**).

Furthermore, the *email*-attribute must be unique. We have used **VARCHAR** for both *username* and *email* instead of **CLOB** as both attributes have constraints on them (which **CLOB** is not guaranteed to support). Finally, both *hash* and *salt* have domain **BINARY**(64) as specified by the description. The **Subscriber** schema translates to the following SQL statement:

```
CREATE TABLE subscriber
(
    username VARCHAR(100) NOT NULL,
    number INT NOT NULL,
    email VARCHAR(200) NOT NULL UNIQUE,
    hash CHAR(64) FOR BIT DATA NOT NULL,
    salt CHAR(64) FOR BIT DATA NOT NULL,
    PRIMARY KEY(username, number)
);
```

As **BINARY** is not supported by our version of DB2, we use the DB2-specific **CHAR**(64) **FOR BIT DATA** instead.

Next, we translate the *Friend_Of* relationship. This relationship is a standard many-to-many relationship without any further constraints. Hence, we translate this relationship into the following relational schema:

FriendOf(<u>from_username</u> : **VARCHAR**(100), <u>from_number</u> : **INT**, <u>to_username</u> : **VARCHAR**(100), <u>to_number</u> : **INT**),

in which the pair of attributes (*from_username*, *from_number*) are a foreign key that refer to the primary key of **Subscriber**, and in which the pair of attributes (*to_username*, *to_number*) are a foreign key that refer to the primary key of **Subscriber**. The **FriendOf** schema translates to the following SQL statement:

[5] The Friend_Of relationship.

[6] This is the normal interpretation of a many-to-many relationship.

[7] Several reviews by the same user for the same film are revisions of each other. This is modeled as a weak entity with partial key revision.

[8] We can have reviews without video or text and with both. Hence, all combinations from the ISA hierarchy are allowed.

[9] The reaction ISA-hierarchy only allows two types of reactions: Thread Reactions (reactions on reactions) and Review Reactions (reactions on reviews).

```
CREATE TABLE friendof
(
    from_username VARCHAR(100) NOT NULL,
    from_number INT NOT NULL,
    to_username VARCHAR(100) NOT NULL,
    to_number INT NOT NULL,
    FOREIGN KEY(from_username, from_number) REFERENCES subscriber,
    FOREIGN KEY(to_username, to_number) REFERENCES subscriber,
    PRIMARY KEY(from_username, from_number, to_username, to_number)
);
```

Before we can translate the **Review** entity, which is a weak entity, we need to know what the primary key of the **film** entity is. In our solution for Part 2, we derive that **film** has a three-attribute primary key (title : **VARCHAR**(100), year : **INT**, creator : **INT**).

Next, we translate the **Review** entity. This entity is both a weak entity and the root of an ISA-hierarchy. For the ISA-hierarchy, both the ER-method and the `NULL` method are suitable: both allow for easily expressing reviews that have *optional* video and/or text components and both methods allows us to easily work with review score summaries of all reviews and easily find either text and/or video information for all reviews that have text and/or videos. The OO method is less-suited in this case, as the OO method will result in 4 different tables in which the review score summaries are scattered and changes to any review (e.g., adding or removing a video component) will move the entire review from one table to the other (assuring that even the details of revisions of a single review are scattered over several tables).

Next, we translate the weak entity **Review** using the `NULL` method into the following relational schema:

Review(author_username : **VARCHAR**(100), author_number : **INT**,
film_title : **VARCHAR**(100), film_year : **INT**, film_creator : **INT**,
revision : **INT**, score : **INT**, timestamp : **TIMESTAMP**,
video : **BLOB**$_{\text{(optional)}}$, description : **CLOB**$_{\text{(optional)}}$).

in which the pair of attributes (*author_username*, *author_number*) are a foreign key that refer to the primary key of **Subscriber**, and in which the triple of attributes (*film_title*, *film_year*, *film_creator*) are a foreign key that refer to the primary key of **Film**. We note that the description does not describe how videos are stored and used. Here, our design assumes that video reviews store the video associated with the review (e.g., an MP4 file) as a large binary object. In the case videos are embedded via an external service (e.g., YouTube, Vimeo, ...), then the attribute *video* should be a text-field sufficiently large to store links to the embedded video.

The **Review** schema translates to the following SQL statement:

```
CREATE TABLE review
(
    author_username VARCHAR(100) NOT NULL,
    author_number INT NOT NULL,
    film_title VARCHAR(100) NOT NULL,
    film_year INT NOT NULL,
    film_creator INT NOT NULL,
    revision INT NOT NULL,
    score INT NOT NULL,
```

```
    timestamp TIMESTAMP NOT NULL,
    video BLOB NULL,
    description CLOB NULL,
    FOREIGN KEY(author_username, author_number) REFERENCES subscriber,
    FOREIGN KEY(film_title, film_year, film_creator) REFERENCES film,
    PRIMARY KEY(author_username, author_number,
                film_title, film_year, film_creator, revision)
);
```

Finally, we translate the **Reaction** entity. This entity is the root of an ISA-hierarchy, but the description puts severe restrictions on this hierarchy: each **Reaction** is one of a reaction-on-a-reaction (**ThreadR**) or a reaction-on-a-review (**ReviewR**). Furthermore, these distinct types of reactions participate in distinct relationships.

To be able to limit relationships involving reactions to either **ThreadR** or **ReviewR** reactions, we need separate tables for both **ThreadR** or **ReviewR**. As such, both the ER method and the OO method are suitable, whereas the `NULL` method is not suitable. To prevent the existence of any reaction that is not a **ThreadR** or **ReviewR** reaction, we can choose for the OO method (instead of the ER method). Unfortunately, we cannot correctly express the *On_Reaction* relationship when using the OO method. Hence, we use the ER method.

Next, we translate the entity **Reaction** using the ER method into the following relational schema:

Reaction(<u>id</u> : **INT**, title : **CLOB**, content : **CLOB**,
by_username : **VARCHAR**(100), by_number : **INT**).

Note that each reaction has a single *author*. Hence, we have incorporated the one-to-many relationship *By* into the schema **Reaction** and the pair of attributes (*by_username*, *by_number*) are a foreign key that refer to the primary key of **Subscriber**. The **Reaction** schema translates to the following SQL statement:

```
CREATE TABLE reaction (
    id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    title CLOB NOT NULL,
    content CLOB NOT NULL,
    by_username VARCHAR(100) NOT NULL,
    by_number INT NOT NULL,
    FOREIGN KEY(by_username, by_number) REFERENCES subscriber
);
```

As the unique identifiers *id* have no meaning outside of the database, we have chosen to always automatically generate these identifiers. This will assure that any row added to the database has a unique never-changing identifier. Next, we translate the reaction specialization **ThreadR** into the following relational schema:

ThreadR(<u>id</u> : **INT**, on_id : **INT**),

in which the primary key *id* is also a foreign keys that refers to the primary key of **Reaction**. As each **ThreadR** reaction is a reaction on another reaction, we have incorporated the one-to-many relationship *On_Reaction* into the schema **ThreadR** and the attribute *on_id* is a foreign key that refers to the primary key of **Reaction**. The **ThreadR** schema translates to the following SQL statement:

```
CREATE TABLE threadr (
    id INT NOT NULL PRIMARY KEY REFERENCES reaction,
    on_id INT NOT NULL REFERENCES reaction
);
```

Finally, we translate the reaction specialization **ReviewR** into the following relational schema

ReviewR(id : **INT**, on_author_username : **VARCHAR**(100), on_author_number : **INT**, on_film_title : **VARCHAR**(100), on_film_year : **INT**, on_film_creator : **INT**, on_revision : **INT**),

in which the primary key *id* is also a foreign keys that refers to the primary key of **Reaction**. As each **ReviewR** reaction is a reaction on a review, we have incorporated the one-to-many relationship *On_Review* into the schema **ReviewR** and the tuple of attributes (*on_author_username*, *on_author_number*, *on_film_title*, *on_film_year*, *on_film_creator*, *on_revision*) is a foreign key that refers to the primary key of **Review**. The **ReviewR** schema translates to the following SQL statement:

```
CREATE TABLE reviewr (
    id INT NOT NULL PRIMARY KEY REFERENCES reaction,
    on_author_username VARCHAR(100) NOT NULL,
    on_author_number INT NOT NULL,
    on_film_title VARCHAR(100) NOT NULL,
    on_film_year INT NOT NULL,
    on_film_creator INT NOT NULL,
    on_revision INT NOT NULL,
    FOREIGN KEY(on_author_username, on_author_number,
                on_film_title, on_film_year, on_film_creator, on_revision)
               REFERENCES review
);
```

We note that this translation does *not* capture the following constraint: "each **reaction** is either a **threadr** or a **reviewr** reaction". This constraint cannot be expressed without using a multi-table constraint, however.

Final remarks This is *not* an example of a clean and easy-to-use design. This is especially the case for the **Review** entity: the usage of *huge primary keys* will complicate query writing (e.g., joining reviews with their video and/or text component). uue to this huge primary key, also any table that stores a relationship to reviews requires many attributes to just express this relationship (e.g., as in **ReviewR**). The addition of a unique identifier for each review in the original ER-Diagram would have resulted in much easier-to-use tables.

*Alternative for **Review** (ER method)*

Next, we translate the weak entity **Review** using the ER method into the following relational schema:

Review(author_username : **VARCHAR**(100), author_number : **INT**, film_title : **VARCHAR**(100), film_year : **INT**, film_creator : **INT**, revision : **INT**, score : **INT**, timestamp : **TIMESTAMP**).

in which the pair of attributes (*author_username*, *author_number*) are a foreign key that refer to the primary key of **Subscriber**, and in which the triple of attributes (*film_title*, *film_year*, *film_creator*) are a foreign key that refer to the primary key of **Film**. The **Review** schema translates to the following SQL statement:

```
CREATE TABLE review
(
    author_username VARCHAR(100) NOT NULL,
    author_number INT NOT NULL,
    film_title VARCHAR(100) NOT NULL,
    film_year INT NOT NULL,
    film_creator INT NOT NULL,
    revision INT NOT NULL,
    score INT NOT NULL,
    timestamp TIMESTAMP NOT NULL,
    FOREIGN KEY(author_username, author_number) REFERENCES subscriber,
    FOREIGN KEY(film_title, film_year, film_creator) REFERENCES film,
    PRIMARY KEY(author_username, author_number,
                film_title, film_year, film_creator, revision)
);
```

Next, we translate the two review specializations into the following relational schemas:

VideoReview(author_username : **VARCHAR**(100), author_number : **INT**,
film_title : **VARCHAR**(100), film_year : **INT**, film_creator : **INT**,
revision : **INT**, video : **BLOB**);
TextReview(author_username : **VARCHAR**(100), author_number : **INT**,
film_title : **VARCHAR**(100), film_year : **INT**, film_creator : **INT**,
revision : **INT**, description : **CLOB**).

The primary keys in both **VideoReview** and **TextReview** are also foreign keys that each refer to the primary key of **Review**. We note that the description does not describe how videos are stored and used. Here, our design assumes that video reviews store the video associated with the review (e.g., an MP4 file) as a large binary object. In the case videos are embedded via an external service (e.g., YouTube, Vimeo, …), then the attribute *video* should be a text-field sufficiently large to store links to the embedded video.

The **VideoReview** and **TextReview** schemas translates to the following SQL statements:

```
CREATE TABLE videoreview
(
    author_username VARCHAR(100) NOT NULL,
    author_number INT NOT NULL,
    film_title VARCHAR(100) NOT NULL,
    film_year INT NOT NULL,
    film_creator INT NOT NULL,
    revision INT NOT NULL,
    video BLOB NOT NULL,
    FOREIGN KEY (author_username, author_number,
                 film_title, film_year, film_creator, revision)
```

```
                    REFERENCES review,
    PRIMARY KEY(author_username, author_number,
                    film_title, film_year, film_creator, revision)
);

CREATE TABLE textreview
(
    author_username VARCHAR(100) NOT NULL,
    author_number INT NOT NULL,
    film_title VARCHAR(100) NOT NULL,
    film_year INT NOT NULL,
    film_creator INT NOT NULL,
    revision INT NOT NULL,
    description CLOB NOT NULL,
    FOREIGN KEY (author_username, author_number,
                    film_title, film_year, film_creator, revision)
                    REFERENCES review,
    PRIMARY KEY(author_username, author_number,
                    film_title, film_year, film_creator, revision)
);
```

Part two: The film information

The consultant sketched the following layout of the database maintaining film information, but indicated that some details still need to be figured out:

- A *table* **person**(<u>id</u>, name, birthdate$_{\text{optional}}$).
  This table provides basic information on people that work on films.

  Solution:
  We only need to translate this schema into SQL. The description does not mention an identifier type or whether these identifiers are to be generated. We choose for the simplest approach: automatically generated identifiers:

  ```
  CREATE TABLE person
  (
      id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
      name CLOB NOT NULL,
      birthdate DATE NULL
  );
  ```

- A *table* **film**(<u>title</u>, <u>year</u>, <u>creator</u>, duration, budget).
  This table represent films. The consultant figured out that films can have the same title and be released in the same year (e.g., in 2005 there was a film *Chaos* directed by Tony Giglio and another film *Chaos* directed by David DeFalco—The consultant didn't see either of them, however). To distinguish between films with the same title and made in the same year, the consultant proposed to use the main director (*creator*, which refers to the id column in the **person** table) of the film.

  Solution:
  We only need to translate this schema into SQL. We use integers for duration (e.g., 123 minutes) and decimals for the budget (e.g., 400000.15 Canadian Dollars).

```
CREATE TABLE film
(
    title VARCHAR(100) NOT NULL,
    year INT NOT NULL,
    creator INT NOT NULL REFERENCES person,
    duration INT NOT NULL,
    budget DECIMAL NOT NULL,
    PRIMARY KEY(title, year, creator)
);
```

- A *view* **film_info** to easily retrieve a film (columns title, year, duration, and budget) together with the name of its creator (column creator_name).

  Solution:

  We only need to create the specified view, which can be created using a straightforward join between the film and the creator tables:

```
CREATE VIEW film_info
AS SELECT F.title, F.year, F.duration, F.budget, C.name AS creator_name
   FROM film F, person C
   WHERE F.creator = C.id;
```

- The consultant wants a table **role** that stores, per film, all people that worked on the film and on the role these people had (e.g., actor, writer, director, producer, costume, casting, editor, or makeup). The consultant had difficulties coming up with a proper design for this table, but noted that people can have several roles in the same film.

  Solution:

  The table role models a many-to-many-to-many relations between (1) *films*, (2) *persons*, and (3) their roles. Hence, the table needs to be structured as follows:

  Role(film_title : **VARCHAR**(100), film_year : **INT**, film_creator : **INT**,
  person_id : **INT**, role : **VARCHAR**(100)),

  in which the triple of attributes (*film_title*, *film_year*, *film_creator*) are a foreign key that refer to the primary key of **Film** and the attribute *person_id* is a foreign key that refers to the primary key of **Person**. Note that with this design, a person can have *multiple* roles on each single that person works on! The **Role** schema translates to the following SQL statement:

```
CREATE TABLE role (
    film_title VARCHAR(100) NOT NULL,
    film_year INT NOT NULL,
    film_creator INT NOT NULL,
    person_id INT NOT NULL REFERENCES person,
    role VARCHAR(100) NOT NULL,
    FOREIGN KEY(film_title, film_year, film_creator) REFERENCES film,
    PRIMARY KEY(film_title, film_year, film_creator, person_id, role)
);
```

- A *constraint* that all film creators (mentioned in the **film** table) have the role of director for that film (as recorded in the **role** table).

  Solution:

  This constraint cannot be expressed without using a multi-table constraint.