Mount Royal University

COMP 2521 002

Data Modeling and Query Languages

Assignment Three

Author
Zyrel Arandia
Author
Andrew Bown
Author
Bryce Carson

December 7, 2023

1 Introduction

The DDL for the assignment SQL is summarized in this top-level chunk. The tables are created in the order the referenced chunks are named. The parental chunk has the same name as the table in the database, but child chunks should have descriptive names (if any child chunks exist).

The SQL server used is MariaDB. The ⟨*MariaDB version*⟩ used is contained in the chunk just referenced.

Occasionally, throughout this document, chunk names may be repeated in SQL comments to ensure that the tangled SQL is readable without the literate document.

⟨*MariaDB version*⟩≡
```
-- 10.3.39-MariaDB; as of 2023-12-02T23:11PM on macomydb.mtroyal.ca
SELECT VERSION();
```

Overall, the SQL script to create the databse is outlined.

⟨*A3.sql*⟩≡
⟨*DDL: tables*⟩

The file is not too complex.

2 DDL

⟨*DDL: tables*⟩≡
⟨*USER*⟩
⟨*BOOK*⟩
⟨*AUTHOR*⟩
⟨*BOOKAUTHOR*⟩
⟨*READBOOK*⟩

3 Entities and Relations

The business rules of the database are not complex, and can be stated in few words. The names of the sections that follow state the rules. Further explanation is given in the body text of a section if necessary.

3.1 USER is an entity

MariaDB VARCHAR data types support different character sets; the UTF-8 character set, which is used for international email addresses, has a maximum length of 21,844 characters in a VARCHAR attribute. Given that, email addresses, nick names, and profiles should each be constrained to this limit.

A profile, or biography, is usually longer than a person's name. Email addresses are often names, and with international email addresses, these may be rather long. The length is indeterminate, so rather than truncating names or biographies, the maximum character length is permited for each string.

Any email address that does not contain an `@` character between a "local part" and a "domain part" is invalid.

From the perspective of a software engineer, almost any string is a valid international email address. ASCII-only email addresses are also quite complex! A great video of a presentation on the topic is available here on YouTube (the presentation was given by Dylan Beattie at NDC { Oslo }). More technical information is contained in this archived document from the Universal Acceptance Steering Group of ICANN.

> In some cases it may be useful to assign both an EAI and a legacy address for a mailbox. (See Downgrading, below.) In some cases there may be a straightforward transliteration, such as борис@domain to boris@domain or 李伟 @domain to liwei@domain. In other cases, there may be no natural way to transliterate, and the two names may have no obvious connection.

Client-side and server-side validation of email addresses should be used; prevent malicious actors. Assuming that client-side validation of email addresses has been implemented properly, and that server-side validation has also occurred, what remains is to insert the email address into the database and check that it has an @ sign in the string, at

minimum (or *maximum*, depending on personal engineering perspective).

A globally usable length for variable character columns is useful; it can be updated in one place, if need be, and all usages will reference the same global constant.

⟨*256 UTF-8 VARCHAR*⟩≡
```
VARCHAR(256) CHARACTER SET utf8
```

⟨*USER*⟩≡
```
-- DONE: creating the table works.
CREATE OR REPLACE TABLE USER (
    Email ⟨256 UTF-8 VARCHAR⟩ PRIMARY KEY CHECK (Email LIKE "%@%"),
    DateAdded DATETIME NOT NULL, -- not user modifiable.
    NickName ⟨256 UTF-8 VARCHAR⟩ UNIQUE,
    Profile ⟨256 UTF-8 VARCHAR⟩
);
```

⟨*USER*⟩+≡
```
-- FIXME: You have an error in your SQL syntax; check the manual that corresponds t
CREATE OR REPLACE TRIGGER afterCreateRow_setDateAddedToNow
BEGIN
UPDATE USER
SET DateAdded = CURDATE();
END;

-- Untested
CREATE OR REPLACE TRIGGER afterUpdateRow_preventDateAddedModification
BEGIN
UPDATE USER
SET new.DateAdded = old.DateAdded
END;
```

3.1.1 Change email procedure

The control model for this database states that email, being the primary key, cannot be changed. To change the email address of a user, the user must be deleted (all with all their associated READBOOK records) before creating a new user with the desired email address. It would be nice to encapsulate this behaviour into a function to make it more reliable. When a user of the database attempts to delete a user, some DCL should be used to prevent the deletion, warn the database user, and inform them to use the provided function which will do the work for them.

⟨*USER*⟩+≡
```
/* Warn database clients that deleting users is impossible manually; they must
use the provided function. */
```

Another convenience would be to automatically add the READBOOK relations back to the new user, so that their user data isn't truly lost; this quotation shows what must be done to enable us to delete a user, as no foreign key references can exist before deleting a record. We do not want to reduce referential integrity. If there is a way to store the information temporarily and insert it into the READBOOK table after deleting the old user and creating a new one, that would be nice; this can perhaps be accomplished with a VIEW or a query result.

> When a user is deleted all the READBOOK records associated with the user must also be deleted.

⟨⟩≡
```
/* TODO: implement the behaviour/functionality described in
the paragraphing of this subsubseciton. */
```

3.2 BOOK is an entity

To prevent books from being deleted, some DCL will be necessary.

[NOTE]: Books can never be deleted.

⟨*BOOK*⟩≡
```
CREATE TABLE BOOK (
  BookID INT PRIMARY KEY AUTO_INCREMENT,
  Title ⟨256 UTF-8 VARCHAR⟩ NOT NULL,
  -- Year is additionally constrained by a trigger to be
  -- <= YEAR(CURDATE()).
  Year INT CHECK (Year >= ⟨date of the Kish Tablet⟩),
  NumRaters INT DEFAULT 0,
  Rating DECIMAL(2,1) DEFAULT 0.0
      CHECK (Rating >= 0.0 AND Rating <= 5.0)
);

⟨prevent the deletion of books⟩
```

⟨*BOOK*⟩+≡
```
CREATE OR REPLACE TRIGGER beforeCreateRow_constrainYearToPresent
BEGIN
IF Year > ⟨the current year⟩
SET Year = ⟨the current year⟩
END;
```

Users obviously cannot read books that have not been written yet, so ⟨*the current year*⟩ is used to limit what can be inserted into the database.

⟨*the current year*⟩≡
```
YEAR(CURDATE())
```

The Kish Tablet is believed to date from 3500 BCE. It is likely the oldest confirmed writing known, so any books a user claims to have read should be more recent than this.

⟨*date of the Kish Tablet*⟩≡
```
-3500
```

Further, we must ⟨*prevent the deletion of books*⟩, so we require a new trigger for that.

The following trigger was inspired by code read from MSSQLTips.

⟨*prevent the deletion of books*⟩≡
```
-- See https://www.mssqltips.com/sqlservertip/2711/different
-- -ways-to-make-a-table-read-only-in-a-sql-server-database/
-- for source of inspiration.
CREATE TRIGGER beforeDeleteRow_warnClientImpossibleChange
    BEFORE DELETE ON BOOK INSTEAD OF DELETE AS
    BEGIN
        -- NOP: TODO: implement the trigger; use rollback?
    END;
```

3.3 AUTHOR is an entity

⟨*AUTHOR*⟩≡
```
CREATE TABLE AUTHOR (
    AuthorID INT PRIMARY KEY AUTO_INCREMENT,
    FirstName ⟨256 UTF-8 VARCHAR⟩ NOT NULL,
    MiddleName ⟨256 UTF-8 VARCHAR⟩,
    Lastname ⟨256 UTF-8 VARCHAR⟩
);
```

3.4 BOOK has AUTHOR

⟨*BOOKAUTHOR*⟩≡
```
CREATE TABLE BOOKAUTHOR (
    AuthorID INT,
    BookID INT
);
```

⟨*BOOKAUTHOR*⟩+≡
```
ALTER TABLE BOOKAUTHOR DROP PRIMARY KEY;

-- Composite primary key composed of foreign keys:
ALTER TABLE BOOKAUTHOR ADD CONSTRAINT
    PRIMARY KEY (AuthorID, BookID);

ALTER TABLE BOOKAUTHOR ADD CONSTRAINT
FOREIGN KEY (AuthorID) REFERENCES AUTHOR (AuthorID)
ON DELETE CASCADE;

-- Books cannot be deleted, so do not permit cascading.
ALTER TABLE BOOKAUTHOR ADD CONSTRAINT
FOREIGN KEY (BookID) REFERENCES BOOK (BookID);
```

3.5 USER reads BOOK

Users can read many books, so necessarily the Email address in a READBOOK should not be unique. This will lead to an excessive amount of storage usage, but this is the specification given in the database design (by our instructor).

> When a user is deleted, all the READBOOK records associated with the user must also be deleted.
>
> The Rating and DateRead can bemodified.

Ergo, a trigger should be implemented to delete the records `WHERE READBOOK.Email = USER.Email` whenever the trigger condition is satisfied.

⟨*READBOOK*⟩≡
```
CREATE TABLE READBOOK (
    BookID INT NOT NULL FOREIGN KEY REFERENCES BOOK(BookID),
    ⟨email⟩,
    -- Constaint DateRead to be less than YEAR(CURDATE()).
    DateRead DATE NOT NULL,
    Rating INT(2) NOT NULL ⟨ensure rating valid⟩)
    ENGINE=INNODB;

-- Include the following code
⟨Update ratings in BOOK on changes to READBOOK⟩
```

Ratings must be constrained between one and ten, but this can be accomplished with a `CHECK` rather than a constraint.

⟨*ensure rating valid*⟩≡
```
CHECK (Rating >= 1 AND Rating <= 10)
```

Ray Ferrell explains in the linked blog how to make changes to a foreign key's referent domain propagate (cascade) to the referring (child) table. This eliminates the need to programmatically delete rows in the table when the referent is updated, and instead the SQL server and supporting engines do the work for us.

⟨*email*⟩≡
```
Email ⟨256 UTF-8 VARCHAR⟩ NOT NULL
    FOREIGN KEY REFERENCES USER(Email)
    ON DELETE CASCADE
```

⟨*READBOOK*⟩+≡
```
ALTER TABLE READBOOK ADD CONSTRAINT
PRIMARY KEY (BookID, Email);
```

The only remaining task is to trigger a call to the user-defined `CalculateRating` aggregate function to update `BOOK.Rating` when this table is updated.

⟨*Update ratings in BOOK on changes to READBOOK*⟩≡
```
CREATE TRIGGER afterUpdateRow_recalculateBookRating
    AFTER UPDATE ⟨recalculate rating after operation⟩

CREATE TRIGGER afterInsertRow_recalculateBookRating
AFTER INSERT ⟨recalculate rating after operation⟩

CREATE TRIGGER afterDeleteRow_recalculateBookRating
    AFTER DELETE ⟨recalculate rating after operation⟩

-- Include the following procedure after this code.
⟨Calculate Rating⟩
```

⟨*recalculate rating after operation*⟩≡
```
ON READBOOK FOR EACH ROW CalculateRating();
```

How `CalculateRating()` performs its work is now defined.

⟨*Calculate Rating*⟩≡

```
CREATE OR REPLACE PROCEDURE CalculateRating
    AS BEGIN
    UPDATE BOOK
    SET Rating = (SELECT AVG(Rating)
                  FROM READBOOK
                  WHERE BOOK.BookID = READBOOK.BookID),
    SET NumRaters = (SELECT COUNT(*)
                     FROM READBOOK
                     WHERE BOOK.BookID = READBOOK.BookID)
    WHERE BOOK.BookID = READBOOK.BookID;
    END;
```

VIEWS are SQL Querys that allow a user to easily see the tables and data from the database. The following views are used to check the data in the database.

⟨*VIEWS used to check data*⟩≡

```
/* This view can be used to identify invalid users (users
without an @ in their email) */
CREATE VIEW InvalidUsers AS
SELECT Email, NickName
FROM
    USER
WHERE
    Email NOT LIKE '%@%';


/* This view provides details about books, including
multiple authors if applicable */
CREATE VIEW BookDetails AS
SELECT B.BookID, B.Title, B.Year, B.NumRaters, B.Rating,
       GROUP_CONCAT(A.FirstName, ' ',
                    A.MiddleName, ' ',
                    A.Lastname) AS Authors
FROM
    BOOK B
LEFT JOIN
    BOOKAUTHOR BA ON B.BookID = BA.BookID
LEFT JOIN
    AUTHOR A ON BA.AuthorID = A.AuthorID
GROUP BY
    B.BookID, B.Title, B.Year, B.NumRaters, B.Rating;


-- View to display books with their average ratings
CREATE VIEW BooksAvgRatings AS
SELECT B.BookID, B.Title, AVG(RB.Rating) AS AverageRating
FROM
    BOOK B
LEFT JOIN
    READBOOK RB ON B.BookID = RB.BookID
GROUP BY
```

```
    B.BookID, B.Title;


/* This view provides statistics about users who have read
and rated books */
CREATE VIEW UserBookDetails AS
SELECT U.Email, U.NickName,
       COUNT(RB.BookID) AS TotalBooksRead,
       AVG(RB.Rating) AS AverageRating
FROM
    USER U
LEFT JOIN
    READBOOK RB ON U.Email = RB.Email
GROUP BY
    U.Email, U.NickName;
```