Table of Contents

[1. Create History Table 1](#_Toc56702110)

[2. Prevent Deletion Trigger 2](#_Toc56702111)

[3. Automatic Update Trigger for DVD Shipment 4](#_Toc56702112)

[4. Automatic Update Trigger for DVD Received 6](#_Toc56702113)

[5. Stored Procedure for Adding Title to Rental Queue 7](#_Toc56702114)

[6. Stored Procedure for Deleting Title from Rental Queue 10](#_Toc56702115)

## 1. Create History Table

Evaluate and implement (create) a **History** table for the Netflix database. The history table should keep track of **rental history**. Review the next few questions to see what you might want to include within the history table. Consider denormalization and what might be useful within the history table beyond just the rental information. Your rental history table could be an audit table, or a denormalized design, or a combination of both. Explain your design, the purpose of your history table and include reasoning for denormalization.

CREATE SEQUENCE rental\_history\_sequence

START WITH 10

INCREMENT BY 10

NOCACHE

NOCYCLE;



CREATE TABLE rentalhistory (

rentalhistoryid int PRIMARY KEY,

rentalid int NOT NULL,

rentalrequestdate DATE NOT NULL,

rentalshippeddate DATE,

rentalreturneddate DATE,

dvdid int NOT NULL,

dvdtitle VARCHAR2(32) NOT NULL,

memberid int NOT NULL,

CONSTRAINT RENTALHISTORY\_RENTALID\_FK FOREIGN KEY (rentalid) REFERENCES rental(rentalid),

CONSTRAINT RENTALHISTORY\_DVDID\_FK FOREIGN KEY (dvdid) REFERENCES dvd(dvdid),

CONSTRAINT RENTALHISTORY\_MEMBERID\_FK FOREIGN KEY (memberid) REFERENCES member(memberid));



Description: I designed my table to basically include the RENTAL table in its entirety. Even though each rental transaction has its own rentalid, I included a rentalhistoryid for the transaction as well because I wanted the table to have its own primary key. I decided to denormalize the rental table into the rentalhistory table, as it already required the ship and return date of the DVD. The member id and dvd id are also in the rental history table to connect to the DVD and MEMBER tables, so it made sense to add in the dvd rental request date to avoid having to JOIN it in the future in order to retrieve information that should already be part of the history. The purpose of my history table is create a table that will avoid the need to use many JOINS to put tables together to get the history information desired, thus increasing efficiency.

**For the next questions make sure to document and test your code:**

* In the case of triggers provide screen-shots of before and after of the data (i.e. what the history table looks like before the trigger is fired, and what it looks like after.)
* For question 5 and 6 please test your procedure by inserting a movie at the beginning, middle and the end of the queue. Your screenshots need to show the queue data before the insert/delete, the insert/delete itself, and the data in the queue after showing that the queue position has changed.

## 2. Prevent Deletion Trigger

Implement a **trigger** for this new **rental history** table that prevents deletions from the table using error handling logic.

CREATE TRIGGER rentalhistory\_delete\_trigger

BEFORE DELETE ON rentalhistory

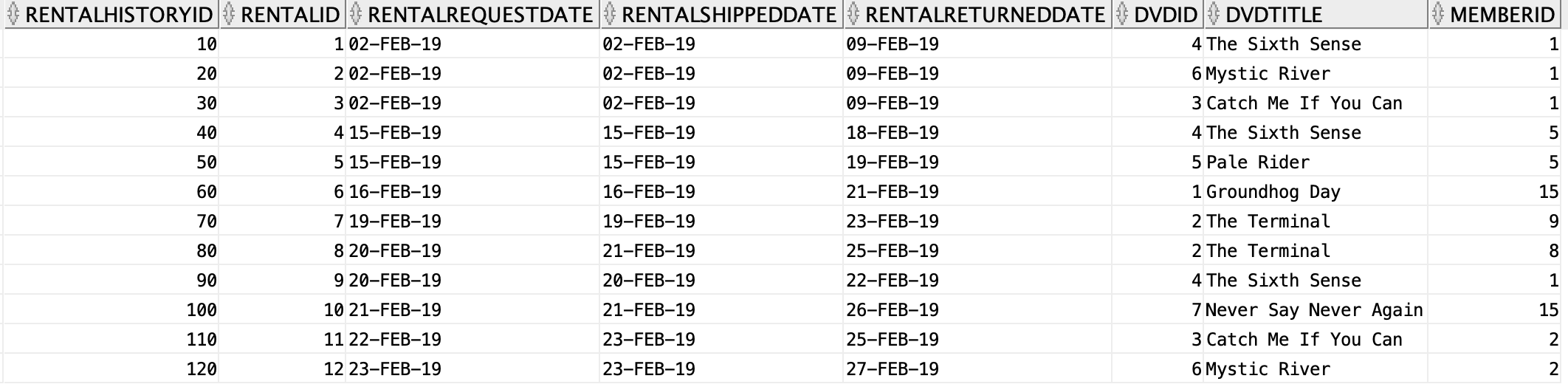
BEGIN

RAISE\_APPLICATION\_ERROR(-20000, 'Records can not be deleted');

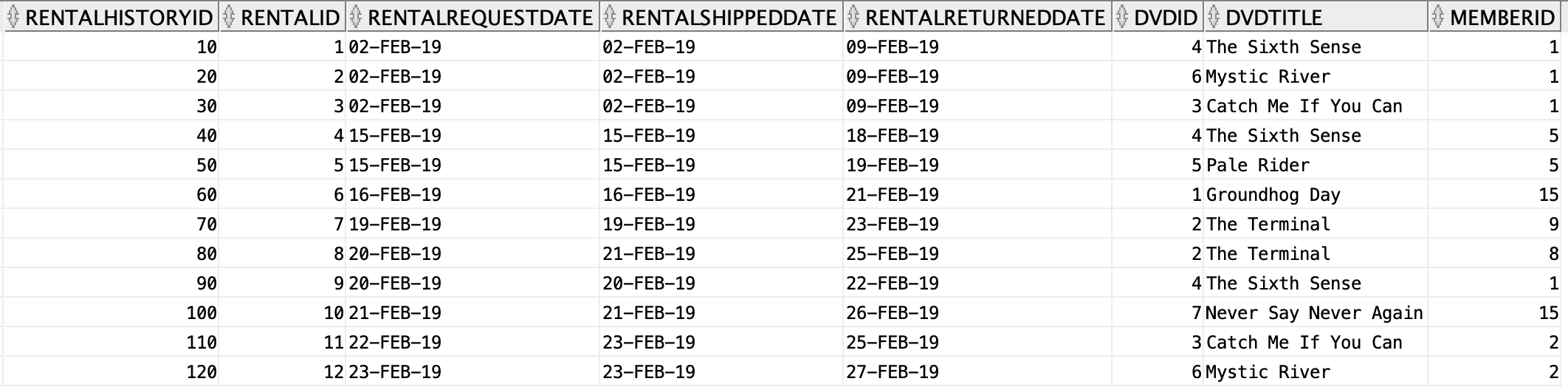
END;



Before Trigger:



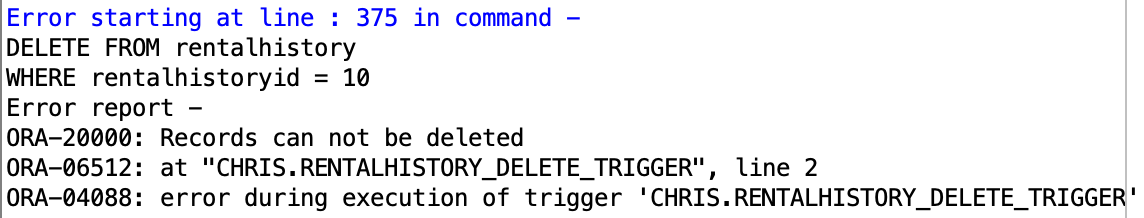
After Trigger:



DELETE

FROM rentalhistory

WHERE rentalhistoryid = 10;



Description: The code creates a trigger named rentalhistory\_delete\_trigger that executes before a DELETE command is run so that an error is raised with a specific message given to the user to let them know the record cannot be deleted. This was tested with a DELETE statement where the first record is attempted to be deleted but, when run, received the ORA-20000 error message.

## 3. Automatic Update Trigger for DVD Shipment

Implement a **trigger** that automatically updates the **rental history** when a DVD is shipped to a customer. Depending on your design of the rental history, this update may be an UPDATE and/or an INSERT.

CREATE TRIGGER rentalhistory\_updateshipdate\_trigger

AFTER UPDATE OF rentalshippeddate ON rental

FOR EACH ROW

BEGIN

UPDATE rentalhistory

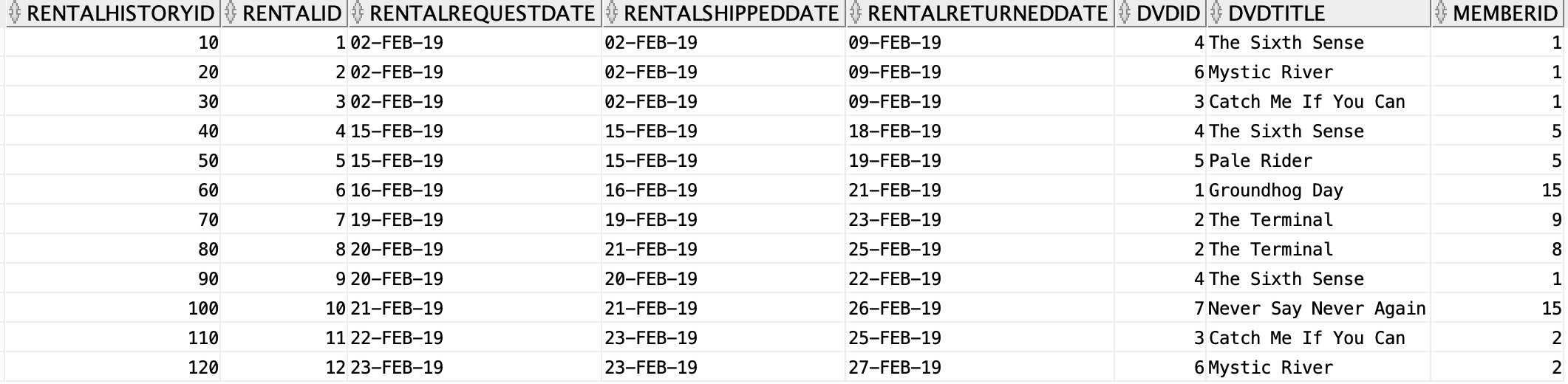
SET rentalhistory.rentalshippeddate = :new.rentalshippeddate

WHERE rentalhistory.rentalid = :new.rentalid;

END;



Before trigger:

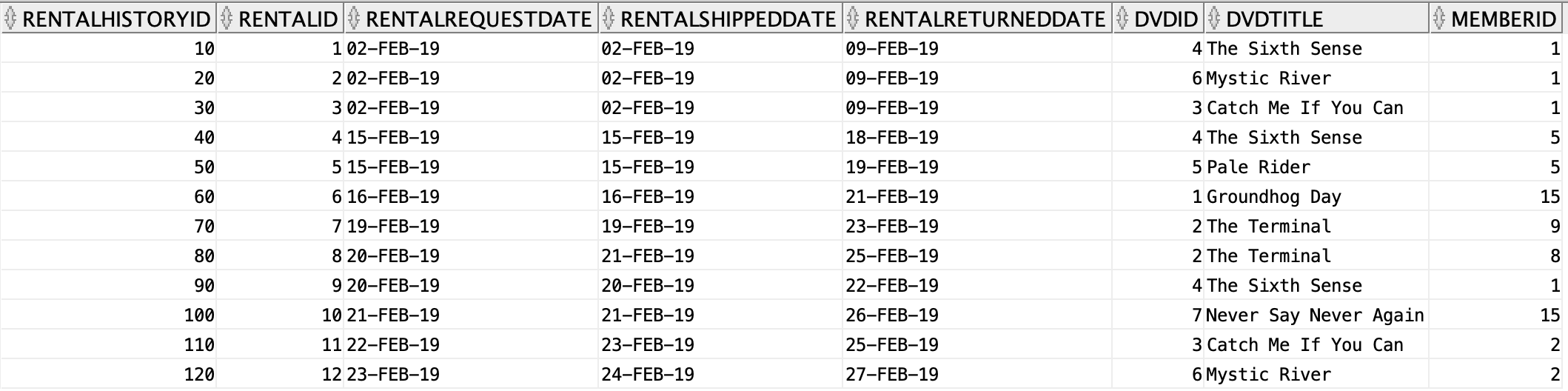


After trigger:

UPDATE rental r

SET r.rentalshippeddate = TO\_DATE('24-FEB-2019', 'DD-MON-YYYY')

WHERE r.rentalid = 12;



Description: The trigger is specifically updating the ship date when there is an UPDATE on the rental history table’s rentalshippeddate column. The INSERT was not included because there are multiple values that are NOT NULL which would require a trigger to be made that inserts all of the new information from the rental table into the rental history table for a new row. The BEGIN/END section simply sets the rental history table’s rentalshipped date value to what it is after there is an update, hence the :new. The test code shows that the rentalid of 12 has a new shippeddate in the rentalhistory table after an update is made on the rental table.

## 4. Automatic Update Trigger for DVD Received

Implement a **trigger** that automatically updates the rental history when a DVD is received from a customer.

CREATE TRIGGER rentalhistory\_updatereturndate\_trigger

AFTER UPDATE OF rentalreturneddate ON rental

FOR EACH ROW

BEGIN

UPDATE rentalhistory

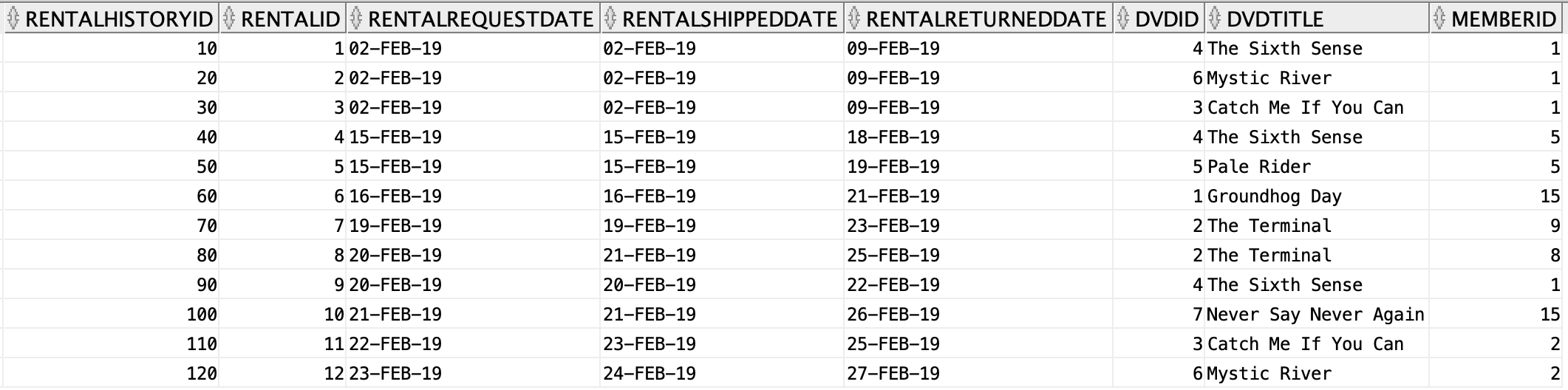
SET rentalhistory.rentalreturneddate = :new.rentalreturneddate

WHERE rentalhistory.rentalid = :new.rentalid;

END;



Before trigger:

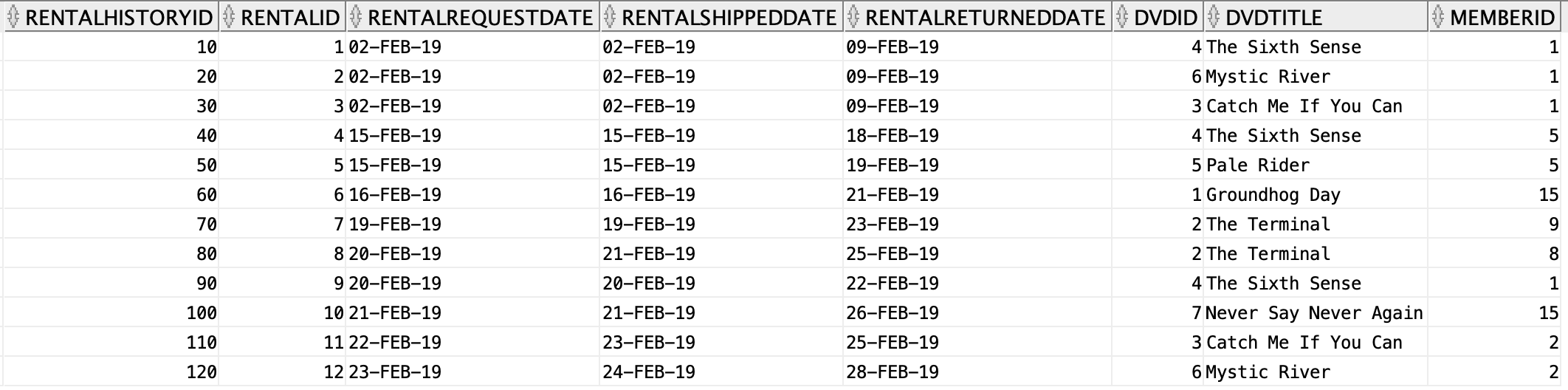


After trigger:

UPDATE rental r

SET r.rentalreturneddate = TO\_DATE('28-FEB-2019', 'DD-MON-YYYY')

WHERE rentalid = 12;



Description: The same principle as question 3, the code updates the rentalreturneddate instead of the rentalshippeddate. It has the data for each row compared to the new rentalreturneddate for its. corresponding rentalid after an UPDATE is made on the rental table.

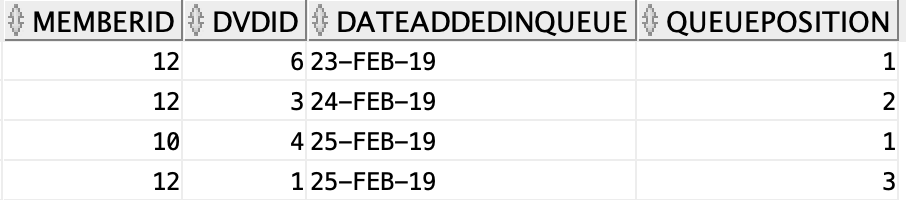
## 5. Stored Procedure for Adding Title to Rental Queue

Implement a **stored procedure** that adds a title to the customer’s movie list (the **Rental Queue** table). This procedure should take as IN parameters the customer ID and movie title ID as well as the location of where the movie is in the queue. The procedure should also make sure that no duplicate titles can be added. You will need to add some error handling in your code. You will also notice that there is no queue position attribute, so you will need to solve that issue.

The stored procedure will need to do some queue position management of the existing movies in the queue. Queue position management allows the customer to rank the movies in their queue. You will have to add this feature to the current schema design.

ALTER TABLE rentalqueue

ADD (queuepositions int);



CREATE OR REPLACE PROCEDURE addMovie (

p\_memberid IN rentalqueue.memberid%TYPE, --customer ID

p\_dvdid IN rentalqueue.dvdid%TYPE, -- movie title ID

p\_dateaddedinqueue IN rentalqueue.dateaddedinqueue%TYPE, -- date

p\_queueposition IN rentalqueue.queueposition%TYPE) --location in Queue

IS

countInRentalQueue NUMBER;

countInQueuePositions NUMBER;

updatingQueuePosition EXCEPTION;

BEGIN

SELECT COUNT(\*) INTO countInQueuePositions

FROM rentalqueue

WHERE memberid = p\_memberid

AND dvdid = p\_dvdid

AND p\_queueposition IN (SELECT queueposition FROM rentalqueue);

SELECT COUNT(\*) INTO countInRentalQueue

FROM rentalqueue

WHERE memberid = p\_memberid

AND dvdid = p\_dvdid

AND p\_queueposition NOT IN (SELECT queueposition FROM rentalqueue);

IF(countInRentalQueue > 0) THEN

RAISE\_APPLICATION\_ERROR(-20001, 'This member''s DVD is already in queue.');

END IF;

IF (countInQueuePositions > 0) THEN

RAISE updatingQueuePosition;

END IF;

INSERT INTO rentalqueue (memberid, dvdid, dateaddedinqueue, queueposition)

VALUES (p\_memberid, p\_dvdid, CURRENT\_DATE, p\_queueposition);

EXCEPTION

WHEN updatingQueuePosition THEN

UPDATE rentalqueue

SET QUEUEPOSITION = QUEUEPOSITION + 1

WHERE memberid = p\_memberid AND queueposition >= p\_queueposition;

UPDATE rentalqueue

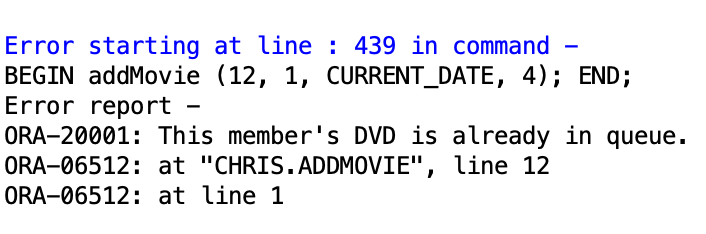
SET queueposition = p\_queueposition

WHERE memberid = p\_memberid AND dvdid = p\_dvdid;

END;

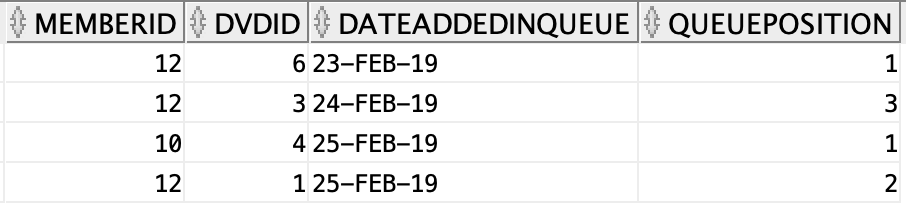
-- Trying to insert duplicate dvd

EXECUTE addMovie (12, 1, CURRENT\_DATE, 4);



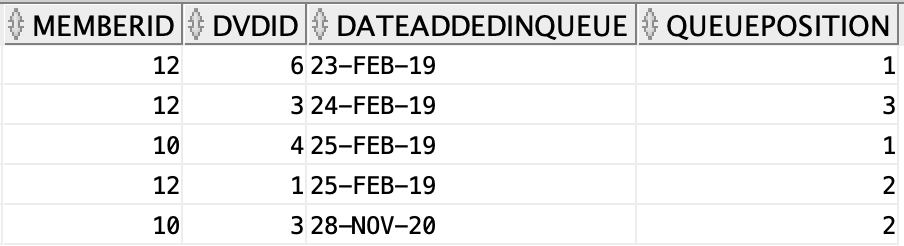
-- Trying to change the position of a movie in queueposition

EXECUTE addmovie(12, 1, CURRENT\_DATE, 2);



-- Inserting dvd normally

EXECUTE addMovie (10, 3, TO\_DATE('28-FEB-2019', 'DD-MON-YYYY'), 2);



Description: The procedure addMovies takes in 4 parameters (p\_memberid, p\_dvdid\_, p\_dateaddedinqueue, p\_queueposition). The IS clause contains variables countInRentalQueue for preventing duplicates, countInQueuePositions for updating queue position, and updatingQueuePosition as the exception to be raised.

When the procedure begins it counts all of the everything into the countInQueuePositions to select anything where the member id, dvdid, and queueposition are already in the rentalqueue table. It then count into the countInRentalQueue variable the rows where the member id, dvdid, are the same as the ones being inserts (aka already in the table) and that the queueposition being inserted is NOT in the table yet. This allows for insertion of dvds normally as well.

The first IF statement raises an application error if the amount of countInRentalQueue is greater than 0, which shows that the member already has that dvd in their queue. The second IF statement raises an exception for when countInQueuePositions is greater than 0, meaning that that member already has that dvd in their queue but is trying to insert it at a different position.

The INSERT statement simply makes it so that dvds can be inserted normally that are neither duplicates for a member, nor ones that are updating queueposition.

The EXCEPTION block updates the queuepositions for that member’s dvds by incrementing their position number by 1, then updating the dvd position they want moved to the inserted value.

Note: Thought the instructions only say “make sure no duplicate titles can be added”, since it has been established previously that there are multiples of DVDs, I wanted to make it so that one member couldn’t put the same DVD in queue twice, but a different member (who has their own queue positions) can have the dvd in queue still. This allows for members to not simply have their queue waiting on another member’s return for a specific dvd. Also, the instructions say “customer ID’ and “movie title ID”. Since neither exists in the tables I interpreted this as memberid and dvdid and left out the title of the films.

## 6. Stored Procedure for Deleting Title from Rental Queue

Write a **stored procedure** that deletes a title from a customer’s movie list (the **Rental Queue** table). This procedure should take as IN parameters the customer ID and movie title ID.

Hint: The stored procedure will need to do some queue position management of the existing movies in the queue building on what you did for the previous requirement.

CREATE OR REPLACE PROCEDURE deleteMovie

(p\_memberid IN rentalqueue.memberid%TYPE, --customer ID

p\_dvdid IN rentalqueue.dvdid%TYPE, -- movie title ID

p\_dateaddedinqueue IN rentalqueue.dateaddedinqueue%TYPE, -- date

p\_queueposition IN rentalqueue.queueposition%TYPE) --location in Queue

IS

countInRentalQueue NUMBER;

countInQueuePositions NUMBER;

updatingQueuePosition EXCEPTION;

BEGIN

SELECT COUNT(\*) INTO countInQueuePositions

FROM rentalqueue

WHERE memberid = p\_memberid;

SELECT COUNT(\*) INTO countInRentalQueue

FROM rentalqueue

WHERE memberid = p\_memberid

AND dvdid = p\_dvdid

AND queueposition = p\_queueposition;

IF (countInRentalQueue = 0) THEN

RAISE\_APPLICATION\_ERROR (-20002, 'This record does not exist.');

END IF;

IF (countInQueuePositions > 1) THEN

RAISE updatingQueuePosition;

END IF;

DELETE FROM rentalqueue

WHERE memberid = p\_memberid

AND dvdid = p\_dvdid

AND queueposition = p\_queueposition;

IF (countInQueuePositions > 0) THEN

UPDATE rentalqueue

SET queueposition = queueposition - 1

WHERE queueposition > p\_queueposition

AND memberid = p\_memberid;

UPDATE rentalqueue

SET queueposition = queueposition + 1

WHERE queueposition < p\_queueposition

AND memberid = p\_memberid;

END IF;

EXCEPTION

WHEN updatingQueuePosition THEN

UPDATE rentalqueue

SET queueposition = 1

WHERE queueposition = 1

AND memberid = p\_memberid;

UPDATE rentalqueue

SET queueposition = queueposition - 1

WHERE queueposition > p\_queueposition

AND memberid = p\_memberid;

DELETE FROM rentalqueue

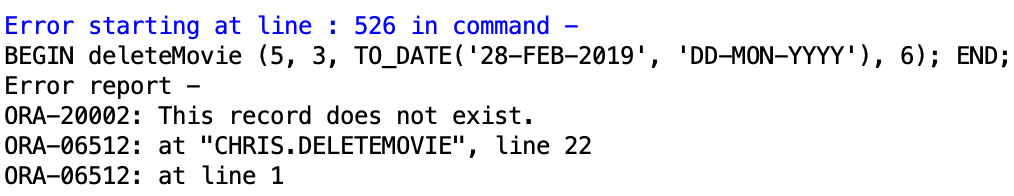
WHERE memberid = p\_memberid

AND dvdid = p\_dvdid

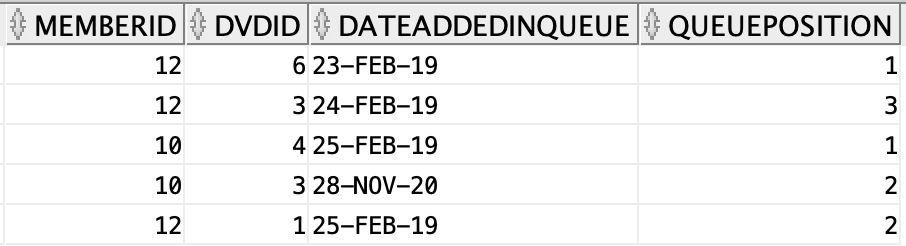
AND queueposition = p\_queueposition;

END;

EXECUTE deleteMovie (5, 3, TO\_DATE('28-FEB-2019', 'DD-MON-YYYY'), 6);-- attempt to delete movie that does not exist in table



Data before procedure:



-- delete movie normally

EXECUTE deleteMovie(12, 1, TO\_DATE('25-FEB-2019', 'DD-MON-YYYY'), 2);

