CS 304 - PART 3: SQL DOC

Rodrigo Alves (Student # 15674112) q1e8

Charles [Tianqi Huang] (Student # 13010087) m4g7

Maria Fung (Student #30448112) e9e8

Jennie Tam (Student #84091065) f5o6

Table Creation

```
//Clean previous tables:
DROP TABLE ReturnItem;
DROP TABLE Return;
DROP TABLE PurchaseItem;
DROP TABLE Purchase;
DROP TABLE Customer;
DROP TABLE HasSong;
DROP TABLE LeadSinger;
DROP TABLE Item;

//Create tables:
CREATE TABLE Item (
upc INTEGER PRIMARY KEY,
title VARCHAR(40) NOT NULL,
type CHAR(3) NOT NULL CHECK (type IN('CD', 'DVD')),
category CHAR(12) NOT NULL CHECK (category IN('rock', 'pop', 'rap', 'country', 'classical', 'new
age', 'instrumental')),
company VARCHAR(40),
year SMALLINT CHECK (year > 1500 AND year < 2500),
price FLOAT CHECK (price >= 0),
stock INTEGER CHECK (stock >= 0) );

CREATE TABLE LeadSinger (
upc INTEGER,
name VARCHAR(40),
PRIMARY KEY (upc, name),
FOREIGN KEY (upc) REFERENCES Item );

CREATE TABLE HasSong (
upc INTEGER,
```

```
title VARCHAR(40),
PRIMARY KEY (upc, title),
FOREIGN KEY (upc) REFERENCES Item );

CREATE TABLE Customer (
cid INTEGER PRIMARY KEY,
password CHAR(32) NOT NULL,
name VARCHAR(40) NOT NULL,
address VARCHAR(40),
phone VARCHAR(20),
UNIQUE (name, password));

CREATE TABLE Purchase(
receiptId INTEGER PRIMARY KEY,
cid INTEGER,
purchaseDate DATE,
card# VARCHAR(20),
expiryDate DATE,
expectedDate DATE,
deliveredDate DATE,
FOREIGN KEY (cid) REFERENCES Customer );

CREATE TABLE PurchaseItem (
receiptId INTEGER,
upc INTEGER,
quantity INTEGER CHECK (quantity > 0),
PRIMARY KEY (receiptId, upc),
FOREIGN KEY (receiptId) REFERENCES Purchase
ON DELETE CASCADE,
FOREIGN KEY (upc) REFERENCES Item );

CREATE TABLE Return (
retid INTEGER PRIMARY KEY,
receiptId INTEGER NOT NULL,
returnDate DATE,
FOREIGN KEY (receiptId) REFERENCES Purchase );

CREATE TABLE ReturnItem (
retid INTEGER,
upc INTEGER,
quantity INTEGER CHECK (quantity > 0),
PRIMARY KEY (retid, upc),
FOREIGN KEY (retid) REFERENCES Return,
```

FOREIGN KEY (upc) REFERENCES Item);

//Sequences to be able to generate customer, return ids, and purchase ids:
DROP SEQUENCE seq_cid;
DROP SEQUENCE seq_receiptId;
DROP SEQUENCE seq_retid;

CREATE SEQUENCE seq_cid MINVALUE 1 START WITH 1 INCREMENT BY 1 CACHE 10;
CREATE SEQUENCE seq_receiptId MINVALUE 1 START WITH 1 INCREMENT BY 1 CACHE 10;
CREATE SEQUENCE seq_retid MINVALUE 1 START WITH 1 INCREMENT BY 1 CACHE 10;

Main Operations

Transactions performed by a clerk [ClerkModel.java]:

- **process a purchase of items in the store:**

  1. First, add the information about the purchase.

  If paying by cash:
  INSERT INTO Purchase (receiptId, cid, purchaseDate, card#, expiryDate, expectedDate, deliveredDate) VALUES (seq_receiptId.nextval, NULL, NULL, NULL, NULL, NULL, NULL)

  If paying by card:
  INSERT INTO Purchase (receiptId, cid, purchaseDate, card#, expiryDate, expectedDate, deliveredDate) VALUES (seq_receiptId.nextval, NULL, NULL, &cardno, TO_DATE(&expiryDate, 'MM-YY'), NULL, NULL)

  2. Then, add to this purchase all items that are being bought:

  INSERT INTO PurchaseItem (receiptId, upc, quantity) VALUES (&receiptId, &upc, &quantity);

  3. Update the purchase information in the Purchase table (The purchaseDate was left NULL until now so the receiptId could be obtained to create purchaseItems (SELECT receiptId FROM Purchase WHERE cid IS NULL AND purchaseDate IS NULL)

  UPDATE Purchase SET purchaseDate = current_date WHERE receiptId = ?"

  4. Once the purchase is completed, update the stock of all items bought:

UPDATE item SET stock = stock - (SELECT quantity FROM PurchaseItem WHERE item.upc = purchaseItem.upc AND receiptId = &receiptId) WHERE EXISTS (SELECT quantity FROM PurchaseItem WHERE item.upc = purchaseItem.upc AND receiptId = &receiptId);

Note: Normally an in-store purchase it may not be necessary to check the stock before making a purchase but for our application there are no physical items being rung up at a store (ie. the stock should always exist if the physical item is in the store) therefore the stock is checked.

SELECT stock FROM Item WHERE upc = ? (then “stock < qty” or not is evaluated. This is not part of the statement so that a set will always return with the amount of stock we have (that way we can warn the user with the amount in the stock currently versus the amount they had asked for (ex: there are only 5 items in stock, you asked for 6))

Note: Items are checked for duplicates before being added to the list. (ex: if the CD “Thriller” was rang up once already, ringing it up again will only increase the quantity of that item to be purchased rather than creating a new record) This is done in the application level as a design choice. Rather than creating records, updating records and deleting records from the table every time a change is made during a transaction, all changes occur in the application level and then when the purchase is actually finalized (customer pays) are the queries made and the tables created.

- **process a return of an item for refund:**

1. First, verify that the refund is within 15 days of purchase and verify that the entered information is correct (ie. purchase and purchase items exist):

SELECT * FROM Purchase P, PurchaseItem I WHERE P.receiptId = &receiptId AND P.receiptId = I.receiptId AND I.upc = &upc AND (current_date - P.purchaseDate) < 16

2. Then check if a return has already occurred in the past for the same item and return the quantity of that item already refunded

SELECT sum(I.quantity) AS \"QUANTITY\" from Return R, ReturnItem I WHERE R.receiptId = &receiptId AND R.retid = I.retid AND I.upc = &upc

3. Obtain the original quantity purchased and verify that the quantity desired for refund is valid. (ie. do not allow refunds for more items than purchased) To do this we first obtain the quantity purchased:

SELECT quantity FROM PurchaseItem WHERE receiptId = ? AND upc = ?

After this we evaluate if (purchasedQty >= (qty + returnedQty)) is true or not.

4. If all is well, we insert the new return record into the Return table and do the same for the ReturnItems

INSERT INTO Return (retid, receiptId, returnDate) VALUES (seq_retid.nextval, &receiptId, NULL)

INSERT INTO ReturnItem (retid, upc, quantity) VALUES (&retid, &upc, &quantity)

5. The return record is updated to include the return date (was left NULL to retrieve the retid: SELECT retid FROM Return WHERE returnDate IS NULL)

UPDATE Return SET returnDate = current_date WHERE retid = &retid

6. Finally, update the stock (assuming the items are in good condition and can be sold again):

UPDATE Item SET stock = stock + &quantity WHERE upc = &upc

Transactions performed by a customer [CustomerModel.java]:

- **registration (if it is the first time accessing the system):**

INSERT INTO Customer(cid, password, name, address, phone) VALUES (seq_cid.nextVal,ORA_HASH('&password'), '&name', '&address', &phone);

Also create a purchase to this customer. This purchase will represent the current cart:

INSERT INTO Purchase VALUES (seq_receiptId.nextVal, &cid, NULL, NULL, NULL, NULL, NULL);

- **logging in:**

SELECT Customer.cid, name, address, phone, receiptId FROM Customer, Purchase WHERE name = '&name' AND password = ORA_HASH('&password') AND Customer.cid = Purchase.cid AND purchaseDate IS NULL;

Note: the receiptId is important to grab, so that we can quickly access the customer cart (represented by that receipt id). The cart is shown as soon as the customer logs in, so we need the receipt id.

- **purchase of items online:**

  1. The user already have a purchase that is created when the user registers or when the user buys something. So, the receiptId should already be in the system since the user logged in. So, when adding items to the cart, just add to that specific receiptId:

  INSERT INTO PurchaseItem VALUES (&receiptId, &upc, &quantity);

  2. When it's time to buy, calculate the total price and the total number of items:

  SELECT SUM(total_price) AS price, SUM(quantity) AS quantity FROM (SELECT quantity, (price * quantity) AS total_price FROM Item, PurchaseItem WHERE Item.upc = PurchaseItem.upc AND PurchaseItem.receiptId = ?)

  3. When buying, update the purchase to add a purchaseDate, the credit card information, and an expectedDate to deliver:

  UPDATE Purchase SET purchaseDate = current_date, expectedDate = current_date + &days, card# = &card, expiryDate = TO_DATE(&expiryDate, 'MM-DD') WHERE receiptId = &receiptId;

  Note: the calculation of days depends on the number of orders in process (SELECT COUNT(*) FROM Purchase WHERE deliveredDate IS NULL) and a constant that represents the maximum number of orders processed in a day.

  4. Once the purchase was completed, update the stock of all items bought:

  UPDATE item SET stock = stock - (SELECT quantity FROM PurchaseItem WHERE item.upc = purchaseItem.upc AND receiptId = &receiptId) WHERE EXISTS (SELECT quantity FROM PurchaseItem WHERE item.upc = purchaseItem.upc AND receiptId = &receiptId);

  5. Now that the purchase has a date, the customer has no cart. So, create a new purchase to represent the new user cart:

  INSERT INTO Purchase VALUES (seq_receiptId.nextVal, &cid, NULL, NULL, NULL, NULL, NULL);

Ps: If any step below fails, delete the Purchase and PurchaseItems inserted.

- **Get items in the cart:**

  1. The cart is the purchase without a purchaseDate. The receiptId of that purchase is saved when the user logs in. So, to get the items in the cart, simply search for that specific receiptId:

  SELECT Item.upc, title, type, category, company, year, price, quantity, (price * quantity) AS total FROM Item, PurchaseItem WHERE Item.upc = PurchaseItem.upc AND PurchaseItem.receiptId = &receiptId;

- **Get well-formatted list of purchased items with total price and quantity:**

  “SELECT Purchase.receiptId AS \"RECEIPT\", total AS \"TOTAL PRICE\", totalQuantity AS \"NUMBER OF ITEMS\", TO_CHAR(purchaseDate, 'dd-MON-yy') AS \"PURCHASE DATE\", substr(card#, length(card#)-4, length(card#)) AS \"CARD - LAST 5 DIGITS\", TO_CHAR(expectedDate, 'dd-MON-yy') AS \"EXPECTED DATE\", TO_CHAR(deliveredDate, 'dd-MON-yy') AS \"DELIVED DATE\" FROM Purchase, (SELECT Purchase.receiptId, SUM(total_price) AS total, SUM(quantity) AS totalQuantity FROM Purchase, (SELECT receiptID, quantity, (price * quantity) AS total_price FROM Item, PurchaseItem WHERE Item.upc = PurchaseItem.upc) I WHERE purchaseDate IS NOT NULL AND I.receiptId = Purchase.receiptId AND cid = &cid GROUP BY (Purchase.receiptId)) P WHERE P.receiptId = Purchase.receiptId ORDER BY purchaseDate");”

Transactions performed by a manager [ManagerModel.java]:

- **add items to the store; this will normally add a number of copies of the same item:**

  1. First, search to see if a specific item to be added does already exist. You can for example, search the item title:

  SELECT upc, title FROM item WHERE title LIKE '%&name%';

  2. If the item you want to add does not exist, first add it to the database with 0 items in stock:

INSERT INTO Item VALUES (&upc, '&title', '&type', '&category', '&company', &year, &price, 0);

3. Then, add more copies to the item you want (given that now you know the upc):

UPDATE Item SET stock = stock + &how_much_to_add WHERE upc = &upc;

- **process the delivery of an order (it just sets the delivery date):**

1. First, get the list of all online deliveries that still wasn't delivered:

SELECT receiptId AS id, cid, TO_CHAR(purchaseDate, 'DD-MON-YY') AS \"PURCHASE DATE\", TO_CHAR(expectedDate, 'DD-MON-YY') AS \"EXPECTED DATE\" FROM Purchase WHERE deliveredDate IS NULL AND expectedDate IS NOT NULL AND purchaseDate IS NOT NULL ORDER BY expectedDate

2. Then, after choosing one purchase, we update its deliveredDate:

UPDATE Purchase SET deliveredDate = TO_DATE(&date, 'YY-MM-DD') WHERE receiptId = &receiptId

- **daily sales report:**

1. First, we get the list of items sold in the day. We make it into a view for use later:

CREATE OR REPLACE VIEW daily_sales AS
SELECT upc, category, price, units, (price*units) as total_value FROM item NATURAL JOIN (SELECT upc, SUM(quantity) AS units FROM purchaseItem, purchase WHERE purchaseItem.receiptId = purchase.receiptId AND TO_CHAR(purchaseDate, 'YY-MM-DD') = '&date' GROUP BY upc) WHERE units > 0 ORDER BY category;

2. We display the view created. It's already ordered by each category:

SELECT upc, category, concat('$', price) AS price, units as \"UNITS SOLD\", concat('$', total_value) as \"PROFIT\" FROM daily_sales

3. Then, we get the total of items from each category:

SELECT category, SUM(units) as \"UNITS SOLD\", concat('$', SUM(total_value)) AS \"PROFIT\" FROM daily_sales GROUP BY category

4. Finally, we get the total of daily sales:

SELECT SUM(total_value) AS total_value, SUM(units) AS total_units FROM daily_sales;

- **top selling items:**

1. Let's use the view created to generate the daily sales report. So, the date was entered when replacing the view. Then, we just need the number of items in the ranking (&n):

```
SELECT ROWNUM as rank, title, company, stock, units as sold FROM item NATURAL JOIN
(SELECT upc, units FROM daily_sales ORDER BY units DESC) WHERE ROWNUM <= &n
```

Default Insertions

To see the default insertions used to test the application, check the file "insert_values.sql" together with this document.