Trading Card Game Database Report

Drew Meseck

4/23/2020

Database Management Systems

Professor Owrang

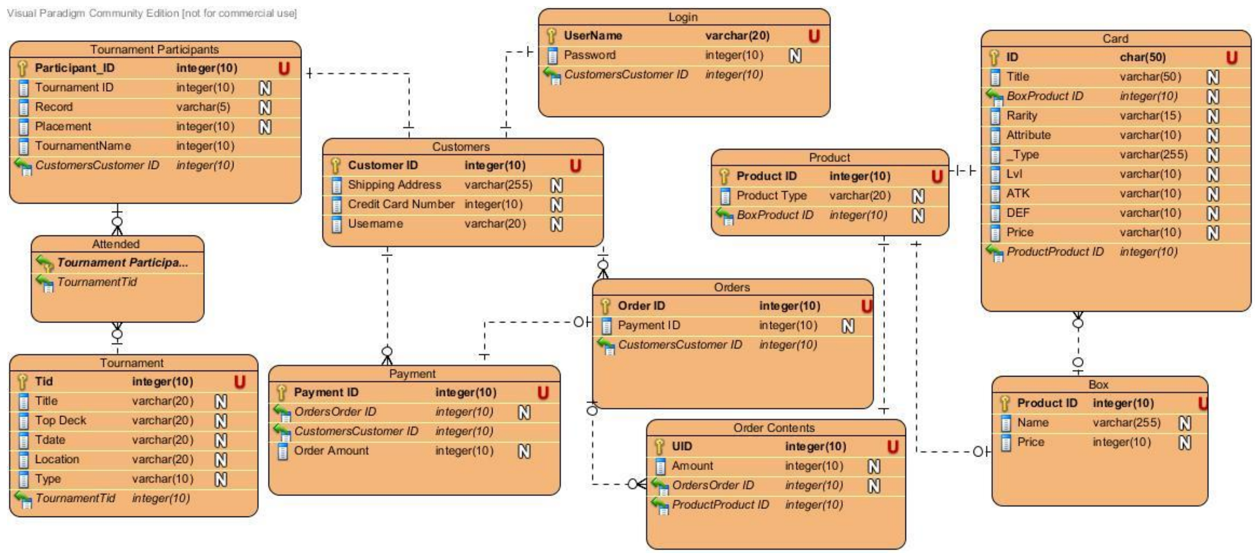
Introduction

The purpose of this database is to simulate a full trading card game online store that tracks not only products and customers, but competitive tournament events as well. These sites (like tcgplayer.com) usually have multiple databases that use relations related to the specific mechanics of games in order to create unique quarriable features. This requires specific attributes that entities possess, and in this way is different from a typical retail database. Since there are restrictions on products, their specific types, and their groupings the design for such a database creates unique challenges to stores like this. Since there is a massive amount of variability in specific products as well it creates the need for optimization in order to make navigating the data efficient. Some example entities are defined below:

- Card
    - A single card, can have many types depending on its mechanical function within the game, each of these different types has different attributes that can affect it, but all are cards
- Set
    - A set contains several cards that are all released together. A pack of trading cards usually belongs to a larger set of cards, which is a pool of cards it can be from. This segments the randomness of actually buying the packs and for consumers makes the process more targeted.
- Tournament
    - A tournament is a competitive event where many users can compete with the products they have purchased.
- Customers
    - Any store or business needs customers, but an online store must carry far more information about consumers (both because of the logistical shipping process as well as the opportunity to provide consumer benefits through stored data). This makes identifying consumers by their login information a necessity. Further relations, like purchase history, can then be used to make suggested purchases as well.

The full description of the database is provided in figure 1, detailing the full ER-diagram of the database.

Figure 1. ER-Diagram for the database including all entities, attributes, and relation types



There are multiple constraints that can be imposed on this database, but most revolve around the payment system. For instance, every order must have a corresponding payment and vice versa in order to ensure that consumers are protected from double charging cards. Furthermore, there are restrictions on the amount any single purchase can be. These restrictions, as described later, will be handled through triggers to either maintain the referential integrity of a foreign key, or to satisfy these domain restrictions. Further restrictions can also be defined with regards to tournaments where every tournament can only have one participant for each placement value, and that every tournament must have a unique tournament ID so that participants of that tournament may be referenced.

Schema and Normal Form Analysis

Dependencies

Card:

ID => Title, Box, Rarity, Attribute, _Type, Lvl, ATK, DEF, Price

For the card table, all attributes are fully dependent on the ID attribute, which is the primary key, and there are no transitive dependencies and is therefore in 3NF.

Box:

Box_ID = > Name, Price

For the box table, all attributes are fully dependent on the Box_ID attribute, which is the primary key, and there are no transitive dependencies or partial dependencies and is also in 3NF

Product:

Product_ID => _Type

For the product table, the only attribute is type, and is therefore fully dependent on the key and has no transitive dependencies. However, product ID is a super key in this case, and therefore the table is in BCNF.

Orders:

Order_ID => Payment_ID, Customer_ID

For orders, both payment identification and customer identification are foreign keys bundled together to represent some unique combination with an order allowing customers to have multiple orders and payments. There are no partial or transitive dependencies in this table. Orders is in 3NF.

Order Contents:

UID => Order_ID, Product_ID, Amount

Order contents was created in order to normalize the orders table which now has no multi-valued dependencies. This table uses a combination of foreign keys from the order and product tables to reference a product that has been ordered and its price. There is here a partial dependency that cannot be further reduced due to referential integrity restrictions making this table 1NF only.

Payments:

Payment_ID => Order_ID, Customer_ID, Amount

While there are no partial dependencies in payments, there is a transitive dependency through customer, however, for security and encryption purposes this could not be reduced further as payments must also be securely associated with a user separately from an order, making this table 2NF due to domain restrictions.

Customers:

Customer_ID => Address, CCN, Username

Customers has no partial or transitive dependencies since each attribute is dependent entirely on the single key putting this table in 3NF.

Login:

Username => Password

The secure nature of logins requires that there be one dependency related to the username which is the password. Username here is a super key making login a BCNF dependency.

Tournament:

Tid => Title, Top_Deck, Tdate, Location, _Type

Tournament is another simple table where each of the attributes here is fully dependent on the key, but that key is not a super key. Therefore, this table is in 3NF

Tournament Participants:

Participant_ID => Tid, Record, Placement, Customer_ID

Tournament Participants are also in 3NF as it only inherits foreign keys from two separate relations that in the context of the table are fully dependent of the Participant ID key.

Query Results Library

Insert Queries

There were a variety of insert queries used in this database. To populate most of the tables, explicit values were passed to insert statements to ensure relations behaved as expected. The notable insert statements are those in the card_inserts.sql file as well as the population of the box table in product_inserts.sql and finally the payments inserts in misc_inserts.sql. The latter of these two utilized select statements from one and multiple tables respectively in order to populate their contents. For the card inserts, a python program was written to generate insert statements to insert the large number of card instances. This program can be found in the Card-Insert-Generator folder included in this project.

While the insert queries provided no explicit outputs as they only insert data into the tables, sample syntax for these inserts are included here:

Card Insert:

```
insert into card values( 'CRMS-EN096', 'Ido the Supreme Magical Force', 'CRMS', 'Effect Monster', 'Secret Rare', 'Dark', 'Fiend', '6', '2200', '800', '36');
```

Box Insert (Using select with one table):

```
insert ignore into box (select card.box, "Box", 69
	from card);
```

Payment Insert (Using Select with multiple tables):

```
insert into Payments (select Orders.Payment_ID, Orders.Order_ID, Orders.Customer_ID, 0 from Orders);
```

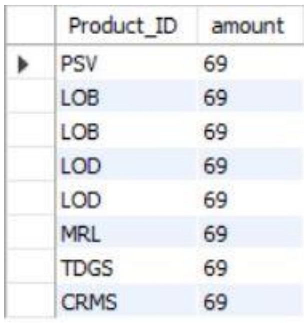
Selection Queries

All selection queries can be found within the Required Queries.sql file and are numbered to reflect their figure output below:

Selection Query 1

In: select order_contents.Product_ID, order_contents.amount from order_contents where order_contents.amount > 50;

Out:

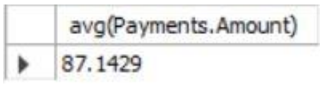


Product_ID	amount
PSV	69
LOB	69
LOB	69
LOD	69
LOD	69
MRL	69
TDGS	69
CRMS	69

Selection Query 2

In: select avg(Payments.Amount) from Payments;

Out:

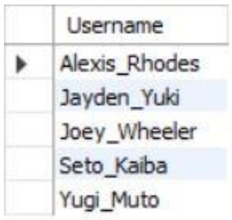


avg(Payments.Amount)
87.1429

Selection Query 3

In: select Username from Customers order by Username;

Out:



Username
Alexis_Rhodes
Jayden_Yuki
Joey_Wheeler
Seto_Kaiba
Yugi_Muto

Selection Query 4

In: select Payments.Customer_ID, sum(Payments.Amount) from Payments

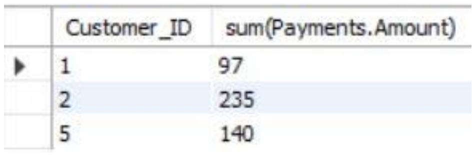
group by

Payments.Customer_ID

having

sum(Payments.Amount) > 70;

Out:



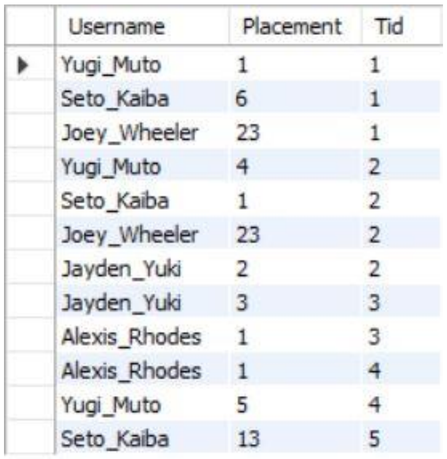
	Customer_ID	sum(Payments.Amount)
▶	1	97
	2	235
	5	140

Selection Query 5

In: select Customers.Username, tournament_participants.Placement, tournament_participants.Tid from Customers

inner join tournament_participants on Customers.Customer_ID = tournament_participants.Customer_ID;

Out:



	Username	Placement	Tid
▶	Yugi_Muto	1	1
	Seto_Kaiba	6	1
	Joey_Wheeler	23	1
	Yugi_Muto	4	2
	Seto_Kaiba	1	2
	Joey_Wheeler	23	2
	Jayden_Yuki	2	2
	Jayden_Yuki	3	3
	Alexis_Rhodes	1	3
	Alexis_Rhodes	1	4
	Yugi_Muto	5	4
	Seto_Kaiba	13	5

Selection Query 6

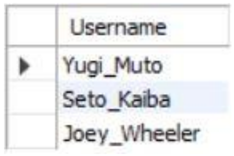
In: select customers.Username from customers

where

customers.Customer_ID in (select tournament_participants.Customer_ID from tournament_participants

where tournament_participants.Tid = 0001);

Out:



	Username
▶	Yugi_Muto
	Seto_Kaiba
	Joey_Wheeler

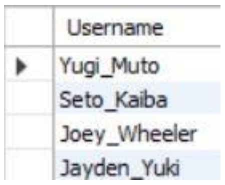
Selection Query 7

In: select customers.Username from customers

where

```
            customers.Customer_ID in (select tournament_participants.Customer_ID from
tournament_participants
                        where tournament_participants.Tid = 0001)
union
select customers.Username from customers
where
            customers.Customer_ID in (select tournament_participants.Customer_ID from
tournament_participants
                        where tournament_participants.Tid = 0002);
```

Out:



```
  Username
▶ Yugi_Muto
  Seto_Kaiba
  Joey_Wheeler
  Jayden_Yuki
```

The description of the purpose of each query is included in a comment in the code files above the corresponding operation.

Update Query

The single update query here is used to update the total price amounts of each payment corresponding to the sum of the amount in each order related to a payment. There is no output for this query, but the syntax of the query is included below:

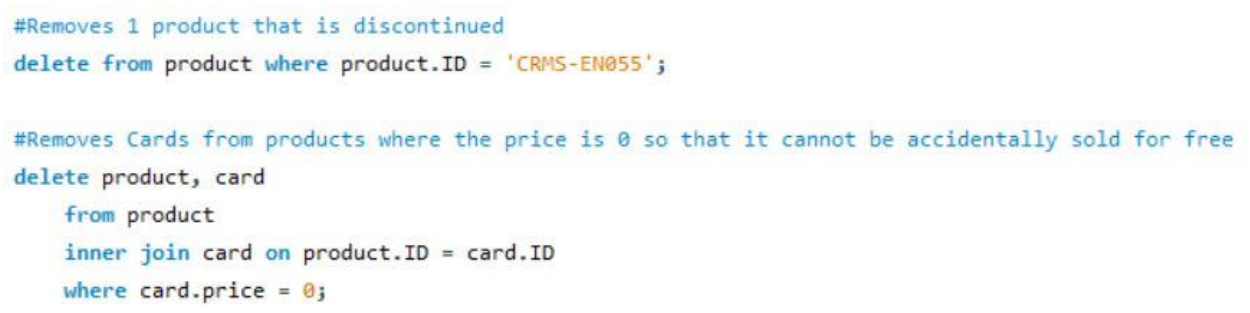
```
UPDATE Payments p
INNER JOIN (
  SELECT order_contents.Order_ID, SUM(order_contents.Amount) as total
  FROM order_contents
  GROUP BY order_contents.Order_ID
) x ON p.Order_ID = x.Order_ID
SET p.Amount = x.total;
```

This update query involves two tables (payments and order contents) joined on the order ID attribute. Payments amount is then updated to reflect the total sum from the order contents table.

Delete Queries

The delete queries utilized for this project are straightforward. The first deletes a specific product from the product table, signifying a product that has been discontinued or has run out of inventory. The second deletes a card from the product table that has no price. This is a domain restriction imposed to ensure no product can be purchased for free due to an insert error.

Figure 2. Delete Queries



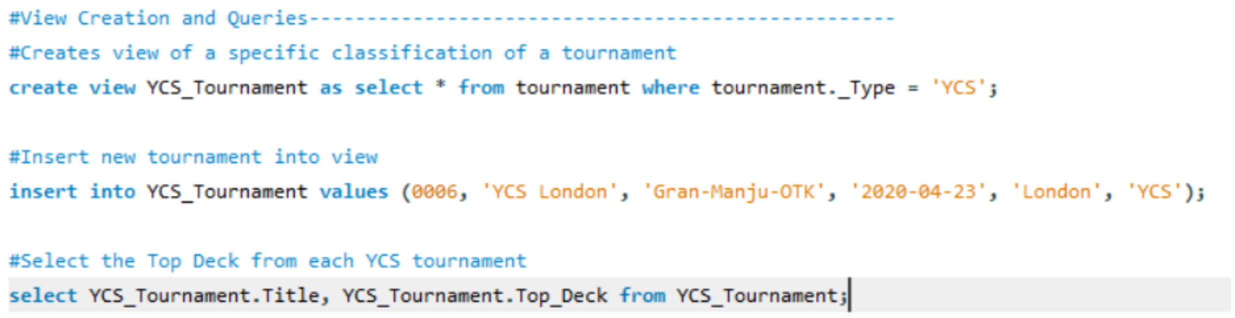
```
#Removes 1 product that is discontinued
delete from product where product.ID = 'CRMS-EN055';

#Removes Cards from products where the price is 0 so that it cannot be accidentally sold for free
delete product, card
    from product
    inner join card on product.ID = card.ID
    where card.price = 0;
```

View Creation and Queries

A view of a specific type of tournament was created in order to do temporary and specific selects involving this specific value.

Figure 3. View Creation and Operations

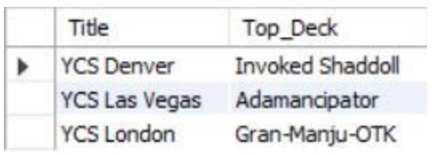


```
#View Creation and Queries--------------------------------------------------
#Creates view of a specific classification of a tournament
create view YCS_Tournament as select * from tournament where tournament._Type = 'YCS';

#Insert new tournament into view
insert into YCS_Tournament values (0006, 'YCS London', 'Gran-Manju-OTK', '2020-04-23', 'London', 'YCS');

#Select the Top Deck from each YCS tournament
select YCS_Tournament.Title, YCS_Tournament.Top_Deck from YCS_Tournament;
```

The select statement highlighted in figure 3 provides the following output:



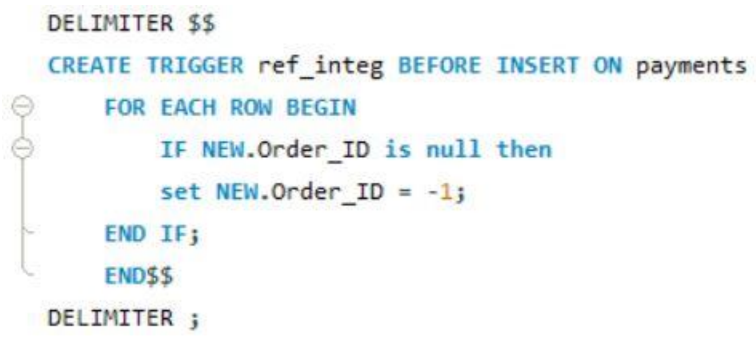
	Title	Top_Deck
▶	YCS Denver	Invoked Shaddoll
	YCS Las Vegas	Adamancipator
	YCS London	Gran-Manju-OTK

Triggers

Referential Integrity Check

The trigger used to maintain referential integrity ensures that any inserted payment has a valid corresponding order ID number. If it does not, this trigger flags it as an invalid payment by setting its ID to negative 1 so it can be handled and not processed as a transaction.

Figure 4. Referential integrity trigger syntax

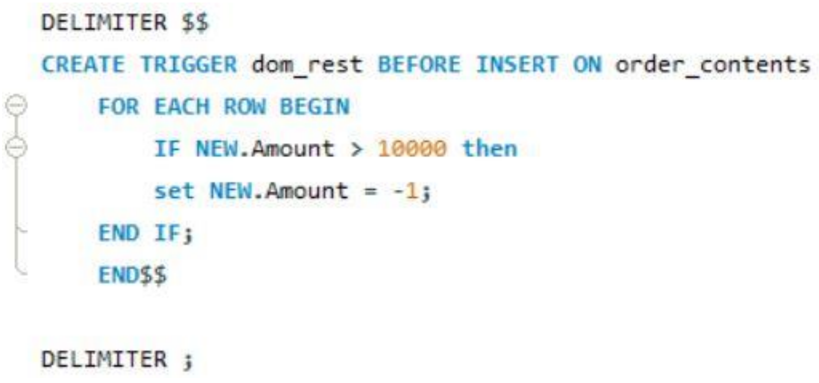


```
DELIMITER $$
CREATE TRIGGER ref_integ BEFORE INSERT ON payments
    FOR EACH ROW BEGIN
        IF NEW.Order_ID is null then
        set NEW.Order_ID = -1;
    END IF;
    END$$
DELIMITER ;
```

Domain Restriction Maintenance

This trigger ensures that the domain restriction limiting the maximum purchase amount placed upon purchases is held. This is important for consumer protection so there is no accidental pricing error or purchasing error. Figure 5 contains the syntax for this trigger.

Figure 5. Domain restriction maintenance trigger syntax



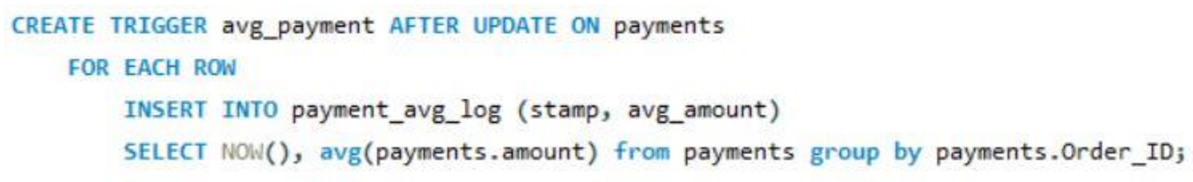
```
DELIMITER $$
CREATE TRIGGER dom_rest BEFORE INSERT ON order_contents
    FOR EACH ROW BEGIN
        IF NEW.Amount > 10000 then
        set NEW.Amount = -1;
    END IF;
    END$$

DELIMITER ;
```

Statistical Gathering Trigger

The trigger used to gather statistics is implemented in order to update the average purchase amount after the purchase table is updated. This is a useful statistic as it shows the volume of purchases and can inform how potential discounts should be targeted based on the magnitude of each purchase. This value along with a timestamp of the time of the operation is written to a log table generated in the schema. The syntax for this trigger is included in figure 6.

Figure 6. Statistics gathering Trigger

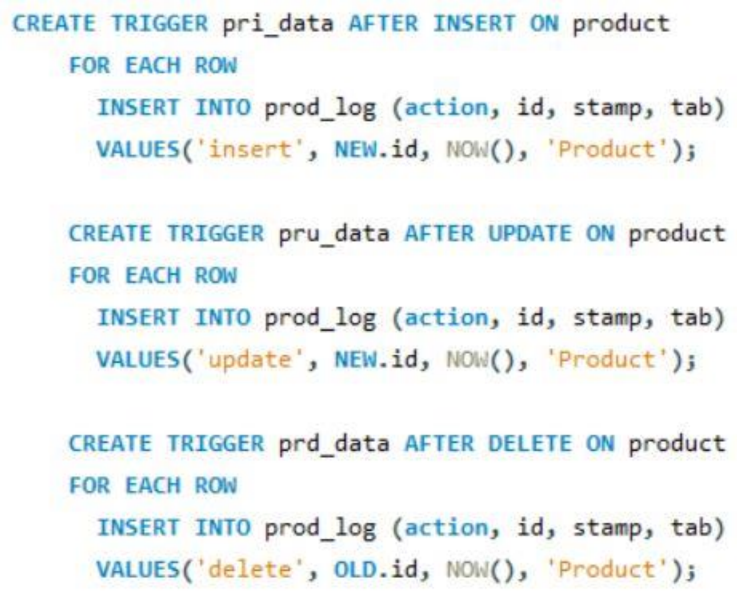


```
CREATE TRIGGER avg_payment AFTER UPDATE ON payments
    FOR EACH ROW
        INSERT INTO payment_avg_log (stamp, avg_amount)
        SELECT NOW(), avg(payments.amount) from payments group by payments.Order_ID;
```

Database Log Creation Through Triggers

Log creation deals with two log tables that are created in the schema. One of these is for products, and the trigger syntax is given in figure 7. This collection of triggers works by checking for some operation on a specific table and logging that action, the ID value of the new or old entry that was operated upon, and the time of that action which are in turn written to the product log.

Figure 7. Product logging triggers for any operation



```
CREATE TRIGGER pri_data AFTER INSERT ON product
    FOR EACH ROW
      INSERT INTO prod_log (action, id, stamp, tab)
      VALUES('insert', NEW.id, NOW(), 'Product');

    CREATE TRIGGER pru_data AFTER UPDATE ON product
    FOR EACH ROW
      INSERT INTO prod_log (action, id, stamp, tab)
      VALUES('update', NEW.id, NOW(), 'Product');

    CREATE TRIGGER prd_data AFTER DELETE ON product
    FOR EACH ROW
      INSERT INTO prod_log (action, id, stamp, tab)
      VALUES('delete', OLD.id, NOW(), 'Product');
```

Other triggers of this variety were created for payments, customers, and orders as well so that this data could also be logged. The same information is logged from the product trigger, but it is logged to a separate table called data log so that the ID type can be consistent between the log tables.

Other Information

All figures in this report are included in the images file under the report file in the main directory. All code is included as SQL scripts and should be run in the order:

1) Schema
2) Triggers
3) Card Inserts
4) Product Inserts
5) Misc Inserts
6) Queries

This order allows one to construct the database as was used for analysis in this report and ensures that no queries are run out of sync.