Final Project-PHFY Database

Ben Hanson

University of Minnesota Crookston

Abstract:

The organization that I've chosen for this project is a company that I created for a previous course, called 'Pond Hockey For You'. This company is very similar to a sporting goods store, such as a Dicks Sporting Goods or Joe's Sporting Goods, only this company solely focuses on selling various types of hockey equipment. The problem that I have solved for the company Pond Hockey For You, is how to keep track of customer information, inventory stock, and more after purchases are made. While many companies have numerous professionals whose jobs are to organize and record this data, small companies like Pond Hockey For You, have no way of doing this essential step. The database management system I have created for this company is a system that records purchase information. Specifically, I have created a database system that records customer and purchase information, including name, address, phone number, email, payment type, item numbers, and other data regarding the specifics of the items which the customers purchase. Due to the fact this company is a very small company that just opened up for sales, I believe providing and maintaining a well organized database management system for these types of data will provide numerous important benefits for this company.

Introduction:

The database management system I have created for Pond Hockey For You is a system that records purchase information. Specifically, I have created a database system that records customer information, such as name, address, phone number, and email, while also recording data regarding the specifics of the items which the customers purchase. This database will have several actors and users which all have very important roles and responsibilities. Database administrators are one type of actor who would work on this database. These actors are responsible for authorizing access and monitoring use of the database. This database will also use multiple types of end users, who will access the database for the purposes of querying, updating, and generating reports. End users who will access this database will be casual end users, parametric end users, and sophisticated end users. Other actors and users of this database will include standalone users, software engineers including system analysts, application programmers, and software engineers.

Database Description of Objects and Relationships:

1. **Screenshot of SQL Server Showing the Database and its Objects**

Shown in the images below are the screenshots showing the PHFY Database and its objects. The database consists of four tables, as well as their rows and columns. The primary keys are underlined, while the foreign keys are italicized.

CUSTOMER

CUSTOMER_ID	*ORDER_ID*	FIRST_NAME	LAST_NAME	PHONE	EMAIL	ADDRESS
1	1319	Joe	Smith	651-795-6417	joesmith10@yahoo.com	8754 Rockaway Lane Mizpah, MN 56660
2	1312	Sam	Johnson	612-752-9721	johnson16@gmail.com	51 Clinton Dr. Forest Lake, MN 55025
3	1575	Sherri	Brown	808-712-3150	sherribrown@outlook.com	8372 E. Vermont Road Randall, MN 56475
4	1590	Sarah	Enstad	612-313-8183	enstadsarah@gmail.com	64 Oak Valley Drive Aitkin, MN 56431

ORDER_INFO

ORDER_ID	*CUSTOMER_ID*	ORDER_DATE	ORDER_STATUS	SHIP_DATE
1312	2	03/10/2020	Delivered	03/17/2020
1575	3	03/28/2020	Shipped	04/02/2020
1319	1	04/12/2020	Received	04/20/2020
1590	4	04/10/2020	Received	04/19/2020
1368	2	03/22/2020	Shipped	04/05/2020

ORDER_ITEMS

ITEM_ID	*ORDER_ID*	QUANTITY	LIST_PRICE	DISCOUNT
52524	*1312*	1	$149.99	-$19.99
67422	*1575*	2	$89.99	0
86345	*1368*	2	$199.99	$9.99
32221	*1319, 1590*	1,1	$79.99	0,0

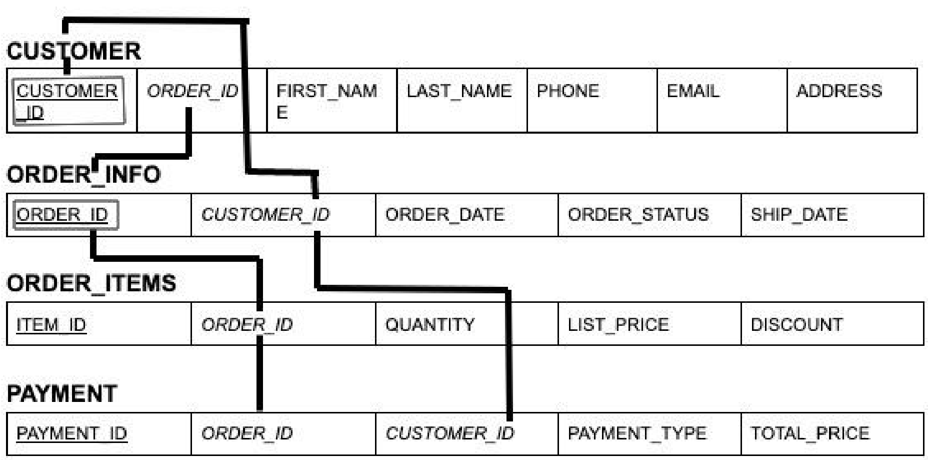
PAYMENT

PAYMENT_ID	*ORDER_ID*	*CUSTOMER_ID*	PAYMENT_TYPE	TOTAL_PRICE
1	1312	2	Credit *6774	$129.99
2	1575	3	Credit *2139	$89.99
3	1319	1	Cash	$79.99
4	1590	4	Credit *4101	$79.99
5	1368	2	Cash	$189.99

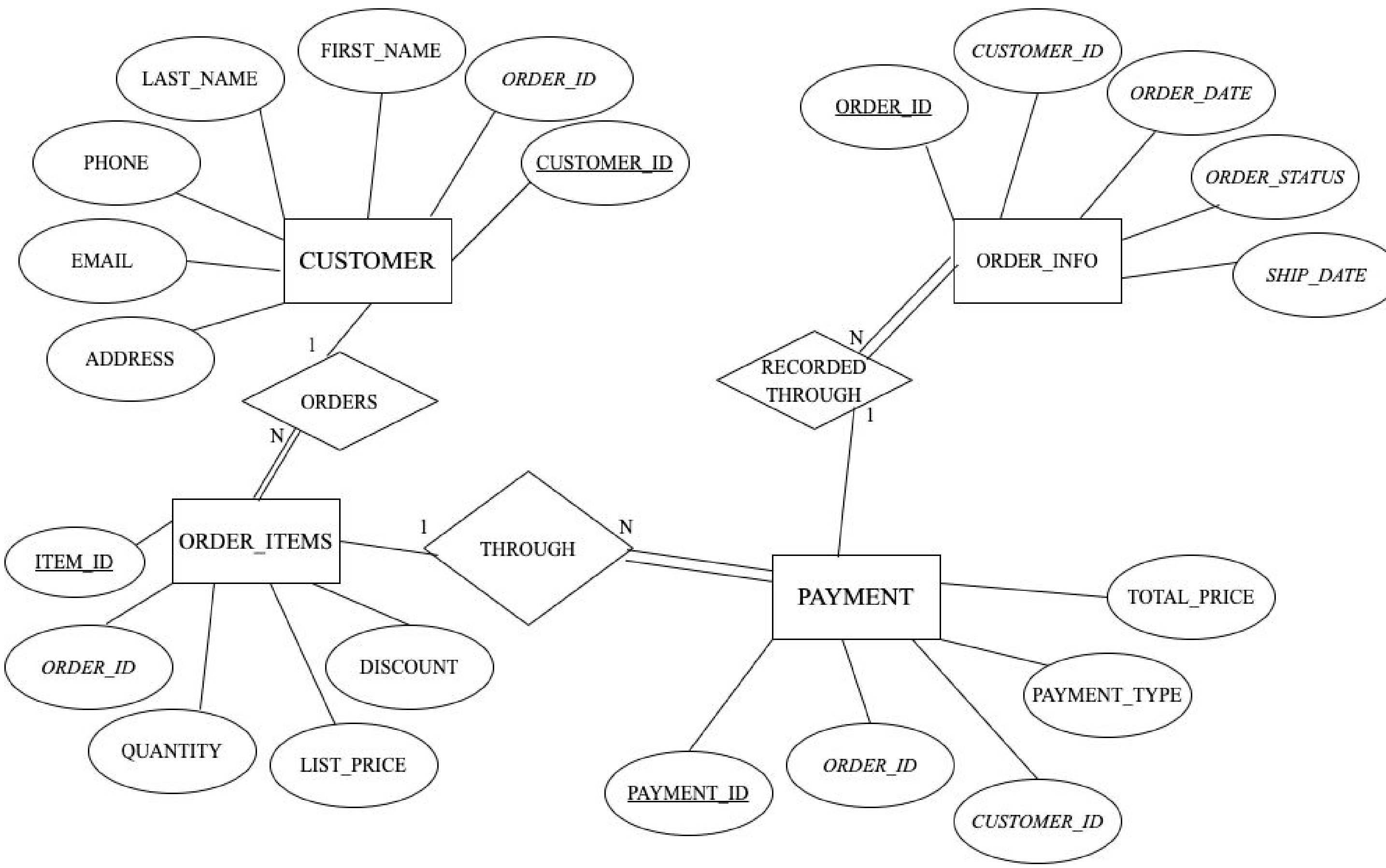
2. **Database Diagram Showing Tables Relationships (FK and PK)**

Shown in the images below are the four tables relationships, through the creation of an Entity Relationship Diagram and a second diagram.

Relationships



Entity Relationship Diagram



3. CREATE SQL Scripts and Screenshot of Database Tables

Shown in the images below are the CREATE SQL scripts that were used to create the tables within the PHFY Database.

Creating and Connecting to the PHFY Database

```
[postgres=# CREATE DATABASE PHFY;
CREATE DATABASE
[postgres=# \c phfy
You are now connected to database "phfy" as user "postgres".
phfy=#
```

Creating the Table 'CUSTOMER'

```
phfy=# CREATE TABLE CUSTOMER  (
CUSTOMER_ID BIGSERIAL NOT NULL PRIMARY KEY,
FOREIGN KEY (ORDER_ID) REFERENCES ORDER (ORDER_ID),
FIRST_NAME VARCHAR(50) NOT NULL,
LAST_NAME VARCHAR(50) NOT NULL,
PHONE VARCHAR(50) NOT NULL,
EMAIL VARCHAR(50) NOT NULL,
ADDRESS VARCHAR(50) NOT NULL );
```

Creating the Table 'ORDER_INFO'

```
[phfy=# CREATE TABLE ORDER_INFO  (
ORDER_ID VARCHAR(50) NOT NULL PRIMARY KEY,
FOREIGN KEY (CUSTOMER_ID) REFERENCES CUSTOMER (CUSTOMER_ID),
ORDER_DATE DATE NOT NULL,
ORDER_STATUS VARCHAR(50) NOT NULL,
SHIP_DATE DATE NOT NULL );
```

Creating the Table 'ORDER_ITEMS'

```
[phfy=# CREATE TABLE ORDER_ITEMS  (
ITEM_ID VARCHAR(50) NOT NULL PRIMARY KEY,
FOREIGN KEY (ORDER_ID) REFERENCES ORDER (ORDER_ID),
QUANTITY VARCHAR(10) NOT NULL,
LIST_PRICE VARCHAR(50) NOT NULL,
DISCOUNT VARCHAR(50) NOT NULL  );
```

Creating the Table 'PAYMENT'

```
[phfy=# CREATE TABLE PAYMENT  (
PAYMENT_ID BIGSERIAL NOT NULL PRIMARY KEY,
FOREIGN KEY (ORDER_ID) REFERENCES ORDER_INFO (ORDER_ID),
FOREIGN KEY (CUSTOMER_ID) REFERENCES CUSTOMER (CUSTOMER_ID),
PAYMENT_TYPE VARCHAR(50) NOT NULL,
TOTAL_PRICE VARCHAR(50) NOT NULL );
```

List of all Tables Created

```
phfy=# \d
                   List of relations
 Schema |          Name             |   Type   |  Owner
--------+---------------------------+----------+----------
 public | customer                  | table    | postgres
 public | customer_customer_id_seq  | sequence | postgres
 public | order_info                | table    | postgres
 public | order_items               | table    | postgres
 public | payment                   | table    | postgres
 public | payment_payment_id_seq    | sequence | postgres
(6 rows)
```

4. INSERT SQL Scripts and Screenshot of Database Tables

Shown in the images below are screenshots of the INSERT SQL scripts where the data was placed into the tables.

INSERT SQL Scripts for Data Being Inserted Into 'CUSTOMER' Table

```
phfy=# INSERT INTO CUSTOMER (
ORDER_ID,
FIRST_NAME,
LAST_NAME,
PHONE,
EMAIL,
ADDRESS )
VALUES ('1319', 'Joe', 'Smith', '651-795-6417', 'joesmith10@yahoo.com
', '8754 Rockaway Lane Mizpah, MN 56660');
INSERT 0 1
phfy=# INSERT INTO CUSTOMER (
ORDER_ID,
FIRST_NAME,
LAST_NAME,
PHONE,
EMAIL,
ADDRESS )
VALUES ('1312', 'Sam', 'Johnson', '612-752-9721', 'johnson16@gmail.co
m', '51 Clinton Dr. Forest Lake, MN 55025');
INSERT 0 1
```

```
phfy=# INSERT INTO CUSTOMER (
ORDER_ID,
FIRST_NAME,
LAST_NAME,
PHONE,
EMAIL,
ADDRESS )
VALUES ('1575', 'Sherri', 'Brown', '808-712-3150', 'sherribrown@outlo
ok.com', '8372 E. Vermont Road Randall, MN 56475');
INSERT 0 1
[phfy=# INSERT INTO CUSTOMER (                                        ]
ORDER_ID,
FIRST_NAME,
LAST_NAME,
PHONE,
EMAIL,
ADDRESS )
VALUES ('1590', 'Sarah', 'Enstad', '612-313-8183', 'enstadsarah@gmail
.com', '64 Oak Valley Drive Aitkin, MN 56431');
INSERT 0 1
```

INSERT SQL Scripts for Data Being Inserted Into 'ORDER_INFO' Table

```
[phfy=# INSERT INTO ORDER_INFO (                                      ]
[phfy(# ORDER_ID,                                                     ]
[phfy(# CUSTOMER_ID,                                                  ]
[phfy(# ORDER_DATE,                                                   ]
[phfy(# ORDER_STATUS,                                                 ]
[phfy(# SHIP_DATE )                                                   ]
[phfy-# VALUES ('1312', '2', DATE '03/10/2020', 'Delivered', DATE '03/]
17/2020');
INSERT 0 1
phfy=# INSERT INTO ORDER_INFO (
ORDER_ID,
CUSTOMER_ID,
ORDER_DATE,
ORDER_STATUS,
SHIP_DATE )
VALUES ('1575', '3', DATE '03/28/2020', 'Shipped', DATE '04/02/2020')
;
INSERT 0 1
```

```
[phfy=# INSERT INTO ORDER_INFO (
ORDER_ID,
CUSTOMER_ID,
ORDER_DATE,
ORDER_STATUS,
SHIP_DATE )
VALUES ('1319', '1', DATE '04/12/2020', 'Received', DATE '04/20/2020'
);
INSERT 0 1
[phfy=# INSERT INTO ORDER_INFO (
ORDER_ID,
CUSTOMER_ID,
ORDER_DATE,
ORDER_STATUS,
SHIP_DATE )
VALUES ('1590', '4', DATE '04/10/2020', 'Received', DATE '04/19/2020'
);
INSERT 0 1
[phfy=# INSERT INTO ORDER_INFO (
ORDER_ID,
CUSTOMER_ID,
ORDER_DATE,
ORDER_STATUS,
SHIP_DATE )
VALUES ('1368', '2', DATE '03/22/2020', 'Shipped', DATE '04/05/2020')
;
INSERT 0 1
```

INSERT SQL Scripts for Data Being Inserted Into ‘ORDER_ITEMS’ Table

```
[phfy=# INSERT INTO ORDER_ITEMS (
[phfy(# ITEM_ID,
[phfy(# ORDER_ID,
[phfy(# QUANTITY,
[phfy(# LIST_PRICE,
[phfy(# DISCOUNT )
[phfy-# VALUES ('52524', '1312', '1', '$149.99','$19.99' );
INSERT 0 1
[phfy=# INSERT INTO ORDER_ITEMS (
ITEM_ID,
ORDER_ID,
QUANTITY,
LIST_PRICE,
DISCOUNT )
VALUES ('67422', '1575', '2', '$89.99','$0' );
INSERT 0 1
[phfy=# INSERT INTO ORDER_ITEMS (
ITEM_ID,
ORDER_ID,
QUANTITY,
LIST_PRICE,
DISCOUNT )
VALUES ('86345', '1368', '2', '$199.99','$9.99' );
INSERT 0 1
[phfy=# INSERT INTO ORDER_ITEMS (
ITEM_ID,
ORDER_ID,
QUANTITY,
LIST_PRICE,
DISCOUNT )
VALUES ('32221', '1319, 1590', '1, 1', '$79.99','$0, $0' );
INSERT 0 1
```

INSERT SQL Scripts for Data Being Inserted Into 'PAYMENT' Table

```
phfy=# INSERT INTO PAYMENT (
phfy(# ORDER_ID,
phfy(# CUSTOMER_ID,
phfy(# PAYMENT_TYPE,
phfy(# TOTAL_PRICE )
phfy-# VALUES ('1312', '2', 'Credit *6774', '$129.99' );
INSERT 0 1
phfy=# INSERT INTO PAYMENT (
ORDER_ID,
CUSTOMER_ID,
PAYMENT_TYPE,
TOTAL_PRICE )
VALUES ('1575', '3', 'Credit *2139', '$89.99' );
INSERT 0 1
phfy=# INSERT INTO PAYMENT (
ORDER_ID,
CUSTOMER_ID,
PAYMENT_TYPE,
TOTAL_PRICE )
VALUES ('1319', '1', 'Cash', '$79.99' );
INSERT 0 1
phfy=# INSERT INTO PAYMENT (
ORDER_ID,
CUSTOMER_ID,
PAYMENT_TYPE,
TOTAL_PRICE )
VALUES ('1590', '4', 'Credit *4101', '$79.99' );
INSERT 0 1
phfy=# INSERT INTO PAYMENT (
ORDER_ID,
CUSTOMER_ID,
PAYMENT_TYPE,
TOTAL_PRICE )
VALUES ('1368', '2', 'Cash', '$189.99' );
INSERT 0 1
phfy=#
```

5. SELECT * FROM TABLE_NAME SQL Scripts and Screenshot of Contents of the Tables

Shown in the images below are the SELECT * FROM TABLE_NAME SQL Scripts that were used to display the contents of the four tables.

SELECT * FROM CUSTOMER Statement Displaying Table

phfy=# SELECT * FROM CUSTOMER;

customer_id	order_id	first_name	last_name	phone	email	address
1	1319	Joe	Smith	651-795-6417	joesmith10@yahoo.com	8754 Rockaway Lane Mizpah, MN 56660
2	1312	Sam	Johnson	612-752-9721	johnson16@gmail.com	51 Clinton Dr. Forest Lake, MN 55025
3	1575	Sherri	Brown	808-712-3150	sherribrown@outlook.com	8372 E. Vermont Road Randall, MN 56475
4	1590	Sarah	Enstad	612-313-8183	enstadsarah@gmail.com	64 Oak Valley Drive Aitkin, MN 56431

(4 rows)

SELECT * FROM ORDER_INFO Statement Displaying Table

phfy=# SELECT * FROM ORDER_INFO;

order_id	customer_id	order_date	order_status	ship_date
1312	2	2020-03-10	Delivered	2020-03-17
1575	3	2020-03-28	Shipped	2020-04-02
1319	1	2020-04-12	Received	2020-04-20
1590	4	2020-04-10	Received	2020-04-19
1368	2	2020-03-22	Shipped	2020-04-05

(5 rows)

SELECT * FROM ORDER_ITEMS Statement Displaying Table

phfy=# SELECT * FROM ORDER_ITEMS;

item_id	order_id	quantity	list_price	discount
52524	1312	1	$149.99	$19.99
67422	1575	2	$89.99	$0
86345	1368	2	$199.99	$9.99
32221	1319, 1590	1, 1	$79.99	$0, $0

(4 rows)

SELECT * FROM PAYMENT Statement Displaying Table

phfy=# SELECT * FROM PAYMENT;

payment_id	order_id	customer_id	payment_type	total_price
1	1312	2	Credit *6774	$129.99
2	1575	3	Credit *2139	$89.99
3	1319	1	Cash	$79.99
4	1590	4	Credit *4101	$79.99
5	1368	2	Cash	$189.99

(5 rows)

Detailed Queries and Screenshots:

Shown below are written SELECT SQL scripts utilizing many different features in the queries. Below are statements of what the query is to return, the script, and a screenshot of the results.

1. **COUNT with GROUP BY and without GROUP BY**

1A. With GROUP BY.

This query will return a row across specified column values. I will be characterizing the data under various groupings, finding the exact number of orders that have been delivered, shipped, and received. The specific script goes as follows:

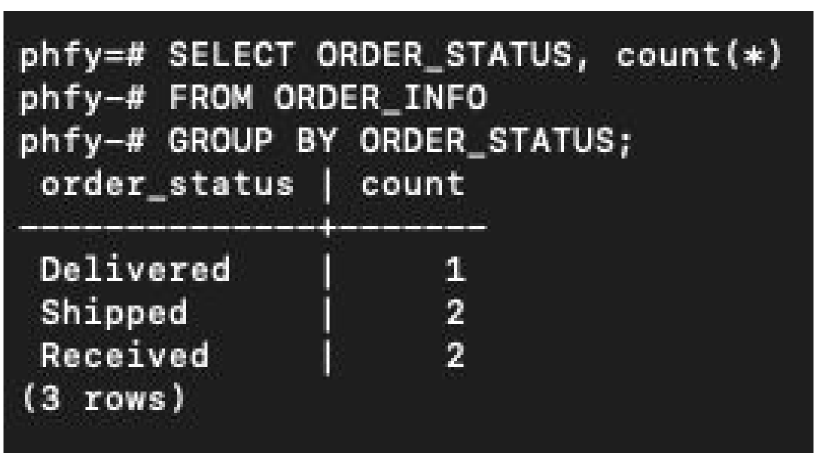
SELECT ORDER_STATUS, count(*)

FROM ORDER_INFO

GROUP BY ORDER_STATUS;

Shown in the image below is the results of this query.

COUNT with GROUP BY Result



1B. Without GROUP BY.

This query will also return a row across specified column values. I will be characterizing the data under various groupings, finding the exact number of orders that have been delivered, shipped, and received. The specific script goes as follows:

SELECT ORDER_STATUS, count(*) OVER() AS COUNT

FROM ORDER_INFO;

Shown in the image below is the results of this query.

COUNT with GROUP BY Result

```
phfy=# SELECT ORDER_STATUS, count(*) OVER() AS COUNT
phfy-# FROM ORDER_INFO;
 order_status | count
--------------+-------
 Delivered    |     5
 Shipped      |     5
 Received     |     5
 Received     |     5
 Shipped      |     5
(5 rows)
```

2. MAX with GROUP BY and without GROUP BY

1A. MAX with GROUP BY.

This query will return the maximum value of an expression, based on a column or a list of separated commas. The specific script goes as follows:

SELECT ORDER_ID, **MAX**(LIST_PRICE)

FROM ORDER_ITEMS

GROUP BY ORDER_ID;

Shown in the image below is the results of this query.

MAX with GROUP BY Result

```
phfy=# SELECT ORDER_ID, MAX( LIST_PRICE)
phfy-# FROM ORDER_ITEMS
phfy-# GROUP BY ORDER_ID;
  order_id  |   max
------------+---------
 1575       | $89.99
 1319, 1590 | $79.99
 1312       | $149.99
 1368       | $199.99
(4 rows)
```

1B. MAX without GROUP BY.

This query will return the maximum value of an expression, based on a column or a list of separated commas. The specific script goes as follows:

SELECT ORDER_ID

FROM ORDER_ITEMS

WHERE ORDER_ID = (**SELECT** MAX(ORDER_ID) **FROM** ORDER_ITEMS);

Shown in the image below is the results of this query.

MAX without GROUP BY Result

```
phfy=# SELECT ORDER_ID
phfy-# FROM ORDER_ITEMS
[phfy-# WHERE ORDER_ID = (SELECT MAX(ORDER_ID) FROM ORDER_ITEMS);
 order_id
----------
 1575
(1 row)
```

3. ORDER BY ASC

This query will return data in a specific order, using the ASC command. The keyword ASC will be used to specify ascending order explicitly. In this example, the results will be the ORDER_INFO table, being placed in ascending order based upon the customer_id. The specific script goes as follows:

SELECT * FROM ORDER_INFO **ORDER BY** CUSTOMER_ID **ASC**;

Shown in the image below is the results of this query.

ORDER BY ASC Result

```
phfy=# SELECT * FROM ORDER_INFO ORDER BY CUSTOMER_ID ASC
[phfy-# ;
 order_id | customer_id | order_date | order_status | ship_date
----------+-------------+------------+--------------+------------
 1319     | 1           | 2020-04-12 | Received     | 2020-04-20
 1312     | 2           | 2020-03-10 | Delivered    | 2020-03-17
 1368     | 2           | 2020-03-22 | Shipped      | 2020-04-05
 1575     | 3           | 2020-03-28 | Shipped      | 2020-04-02
 1590     | 4           | 2020-04-10 | Received     | 2020-04-19
(5 rows)
```

4. ORDER BY DESC

This query will return data in a specific order, using the DESC command. The keyword DESC will be used to specify descending order explicitly. In this example, the results will be the CUSTOMER table, being placed in descending order based upon the customer_id. The specific script goes as follows:

SELECT * FROM PAYMENT **ORDER BY** CUSTOMER_ID **DESC**;

Shown in the image below is the results of this query.

ORDER BY DESC Result

```
phfy=# SELECT * FROM PAYMENT ORDER BY CUSTOMER_ID DESC;
 payment_id | order_id | customer_id | payment_type | total_price
------------+----------+-------------+--------------+-------------
          4 | 1590     | 4           | Credit *4101 | $79.99
          2 | 1575     | 3           | Credit *2139 | $89.99
          1 | 1312     | 2           | Credit *6774 | $129.99
          5 | 1368     | 2           | Cash         | $189.99
          3 | 1319     | 1           | Cash         | $79.99
(5 rows)
```

5. WHERE clause

This query will return data based on the WHERE clause, which will compare a given value with the field value available in the table. In this example, the script will return all records from the ORDER_ITEMS table where there is a discount equal to $0.00. The specific script goes as follows:

SELECT * from ORDER_ITEMS **WHERE** DISCOUNT = '$0' **OR** DISCOUNT = '$0, $0';

Shown in the image below is the results of this query.

WHERE clause Result

```
phfy=# SELECT * from ORDER_ITEMS WHERE DISCOUNT = '$0' OR DISCOUNT =
 '$0, $0';
 item_id |  order_id   | quantity | list_price | discount
---------+-------------+----------+------------+----------
 67422   | 1575        | 2        | $89.99     | $0
 32221   | 1319, 1590  | 1, 1     | $79.99     | $0, $0
(2 rows)
```

6. HAVING clause

This query will return data based on a specified filter conditions for a group of rows. The results in this example will be based on the specified condition, HAVING

COUNT (QUANTITY) > 2. By doing this, the result will be the specific customers who ordered more than two items. The specific script goes as follows:

SELECT COUNT (QUANTITY), ORDER_ID

FROM ORDER_ITEMS

GROUP BY ORDER_ID

HAVING COUNT (QUANTITY) > 2;

Shown in the image below is the results of this query.

HAVING CLAUSE Result

```
phfy=# SELECT COUNT(QUANTITY), ORDER_ID
FROM ORDER_ITEMS
GROUP BY ORDER_ID
HAVING COUNT (QUANTITY) > 2;
 count | order_id
-------+----------
(0 rows)
```

7. INNER JOIN

This query will return data by selecting all rows from both of the participating tables. The INNER JOIN will join two tables according to the matching of some criteria using a comparison operator. The specific script goes as follows:

SELECT ORDER_INFO.ORDER_ID, ORDER_INFO.CUSTOMER_ID,

ORDER_ITEMS.ITEM_ID, ORDER_ITEMS.QUANTITY

FROM ORDER_INFO

INNER JOIN ORDER_ITEMS

ON ORDER_INFO.ORDER_ID = ORDER_ITEMS.ORDER_ID;

Shown in the image below is the results of this query.

INNER JOIN Result

```
[phfy=# SELECT ORDER_INFO.ORDER_ID, ORDER_INFO.CUSTOMER_ID, ORDER_ITEM]
 S.ITEM_ID, ORDER_ITEMS.QUANTITY
[phfy-# FROM ORDER_INFO                                               ]
[phfy-# INNER JOIN ORDER_ITEMS                                        ]
[phfy-# ON ORDER_INFO.ORDER_ID = ORDER_ITEMS.ORDER_ID;                ]
  order_id | customer_id | item_id | quantity
 ----------+-------------+---------+----------
  1312     | 2           | 52524   | 1
  1575     | 3           | 67422   | 2
  1368     | 2           | 86345   | 2
 (3 rows)
```

8. LEFT OUTER JOIN

This query will return data in which two tables are joined, fetching all the matching rows of the two tables where the given expression is true. Also, a LEFT OUTER JOIN will add rows from the first table that do not match any row within the second table.. The specific script goes as follows:

SELECT CUSTOMER.CUSTOMER_ID, CUSTOMER.ORDER_ID,

PAYMENT.TOTAL_PRICE

FROM PAYMENT

LEFT JOIN CUSTOMER

ON CUSTOMER.ORDER_ID=PAYMENT.ORDER_ID

ORDER BY ORDER_ID;

Shown in the image below is the results of this query.

LEFT OUTER JOIN Result

```
[phfy=# SELECT CUSTOMER.CUSTOMER_ID, CUSTOMER.ORDER_ID, PAYMENT.TOTAL_]
 PRICE
[phfy-# FROM PAYMENT                                                  ]
[phfy-# LEFT JOIN CUSTOMER                                            ]
[phfy-# ON CUSTOMER.ORDER_ID=PAYMENT.ORDER_ID                         ]
[phfy-# ORDER BY ORDER_ID;                                            ]
 customer_id | order_id | total_price
-------------+----------+-------------
           2 | 1312     | $129.99
           1 | 1319     | $79.99
           3 | 1575     | $89.99
           4 | 1590     | $79.99
             |          | $189.99
(5 rows)
```

9. RIGHT OUTER JOIN

This query will return records from the right table, or table 2, as well as the matched records from the left table, table 1. The specific script goes as follows:

SELECT ORDER_INFO.SHIP_DATE, CUSTOMER.FIRST_NAME, CUSTOMER.LAST_NAME

FROM ORDER_INFO

RIGHT JOIN CUSTOMER

ON ORDER_INFO.ORDER_ID=CUSTOMER.ORDER_ID

ORDER BY ORDER_INFO.SHIP_DATE;

Shown in the image below is the results of this query.

RIGHT OUTER JOIN Result

```
[phfy=# SELECT CUSTOMER.CUSTOMER_ID, CUSTOMER.ORDER_ID, PAYMENT.TOTAL_]
PRICE
[phfy-# FROM PAYMENT                                                  ]
[phfy-# LEFT JOIN CUSTOMER                                            ]
[phfy-# ON CUSTOMER.ORDER_ID=PAYMENT.ORDER_ID                         ]
[phfy-# ORDER BY ORDER_ID;                                            ]
 customer_id | order_id | total_price
-------------+----------+-------------
           2 | 1312     | $129.99
           1 | 1319     | $79.99
           3 | 1575     | $89.99
           4 | 1590     | $79.99
             |          | $189.99
(5 rows)
```

10. FULL OUTER JOIN

This query will return all the records when there is a match in the left table, table 1, or the right table, table 2, table records. The specific script goes as follows:

SELECT CUSTOMER.EMAIL, CUSTOMER.ADDRESS,

PAYMENT.PAYMENT_TYPE

FROM CUSTOMER

FULL OUTER JOIN PAYMENT

ON CUSTOMER.CUSTOMER_ID=PAYMENT.CUSTOMER_ID

ORDER BY CUSTOMER.EMAIL;

Shown in the image below is the results of this query.

FULL OUTER JOIN Result

```
[phfy=# SELECT CUSTOMER.EMAIL, CUSTOMER.ADDRESS, PAYMENT.PAYMENT_]
TYPE
FROM CUSTOMER
FULL OUTER JOIN PAYMENT
ON CUSTOMER.ORDER_ID=PAYMENT.ORDER_ID
ORDER BY CUSTOMER.EMAIL;
```

```
          email            |                 address                  | payment_type
---------------------------+------------------------------------------+--------------
 enstadsarah@gmail.com     | 64 Oak Valley Drive Aitkin, MN 56431     | Credit *4101
 joesmith10@yahoo.com      | 8754 Rockaway Lane Mizpah, MN 56660      | Cash
 johnson16@gmail.com       | 51 Clinton Dr. Forest Lake, MN 55025     | Credit *6774
 sherribrown@outlook.com   | 8372 E. Vermont Road Randall, MN 56475   | Credit *2139
                           |                                          | Cash
(5 rows)
```

11. CROSS JOIN

This query will return a result set in which the number of rows in the first table is multiplied by the number of rows in the second table, when no WHERE clause is used with the CROSS JOIN. The specific script goes as follows:

SELECT ORDER_INFO.ORDER_DATE, ORDER_INFO.ORDER_ID,

ORDER_ITEMS.LIST_PRICE, ORDER_ITEMS.DISCOUNT

FROM ORDER_INFO

CROSS JOIN ORDER_ITEMS;

Shown in the image below is the results of this query.

CROSS JOIN Result

```
phfy=# SELECT ORDER_INFO.ORDER_DATE, ORDER_INFO.ORDER_ID, ORDER
_ITEMS.LIST_PRICE, ORDER_ITEMS.DISCOUNT
[phfy-# FROM ORDER_INFO                                       ]
[phfy-# CROSS JOIN ORDER_ITEMS;                               ]
 order_date | order_id | list_price | discount
------------+----------+------------+----------
 2020-03-10 | 1312     | $149.99    | $19.99
 2020-03-10 | 1312     | $89.99     | $0
 2020-03-10 | 1312     | $199.99    | $9.99
 2020-03-10 | 1312     | $79.99     | $0, $0
 2020-03-28 | 1575     | $149.99    | $19.99
 2020-03-28 | 1575     | $89.99     | $0
 2020-03-28 | 1575     | $199.99    | $9.99
 2020-03-28 | 1575     | $79.99     | $0, $0
 2020-04-12 | 1319     | $149.99    | $19.99
 2020-04-12 | 1319     | $89.99     | $0
 2020-04-12 | 1319     | $199.99    | $9.99
 2020-04-12 | 1319     | $79.99     | $0, $0
 2020-04-10 | 1590     | $149.99    | $19.99
 2020-04-10 | 1590     | $89.99     | $0
 2020-04-10 | 1590     | $199.99    | $9.99
 2020-04-10 | 1590     | $79.99     | $0, $0
 2020-03-22 | 1368     | $149.99    | $19.99
 2020-03-22 | 1368     | $89.99     | $0
 2020-03-22 | 1368     | $199.99    | $9.99
 2020-03-22 | 1368     | $79.99     | $0, $0
(20 rows)
```

12. WHERE clause with at least one comparison operator (=, <, >, <=, >=, <>)

This query will return data based on a restriction from the WHERE clause, using at least one comparison operator. The specific script goes as follows:

SELECT * FROM ORDER_INFO **WHERE** CUSTOMER_ID = ‘2’;

Shown in the image below is the results of this query.

WHERE CLAUSE With Comparison Operator Result

```
phfy=# SELECT * FROM ORDER_INFO WHERE CUSTOMER_ID = '2';
 order_id | customer_id | order_date | order_status | ship_date
----------+-------------+------------+--------------+-----------
 1312     | 2           | 2020-03-10 | Delivered    | 2020-03-17
 1368     | 2           | 2020-03-22 | Shipped      | 2020-04-05
(2 rows)
```

13. WHERE clause with the LIKE operator (use a wildcard character)

This query will return specified data based on a specified pattern in a column, using a wildcard character such as % or _. The specific script goes as follows:

SELECT * FROM CUSTOMER

WHERE LAST_NAME **LIKE** ‘n%’;

Shown in the image below is the results of this query.

WHERE Clause With Wildcard Character Result

```
phfy=# SELECT * FROM CUSTOMER
WHERE LAST_NAME LIKE '%s%';
```

customer_id	order_id	first_name	last_name	phone	email	address
2	1312	Sam	Johnson	612-752-9721	johnson16@gmail.com	51 Clinton Dr. Forest
4	1590	Sarah	Enstad	612-313-8183	enstadsarah@gmail.com	64 Oak Valley Drive A

(2 rows)

14. WHERE clause with the BETWEEN operator

This query will return data that is between a given range. The specific script goes as follows:

SELECT ORDER_ID, PAYMENT_TYPE, TOTAL_PRICE

FROM PAYMENT

WHERE TOTAL_PRICE **BETWEEN** 80.00 **AND** 200.00;

Shown in the image below is the results of this query.

WHERE Clause With BETWEEN Operator Result

```
[phfy=# SELECT ORDER_ID, PAYMENT_TYPE, TOTAL_PRICE
phfy-# FROM PAYMENT
phfy-# WHERE TOTAL_PRICE BETWEEN 80.00 AND 200.00;
 order_id | payment_type | total_price
----------+--------------+-------------
 1312     | Credit *6774 |      129.99
 1575     | Credit *2139 |       89.99
 1368     | Cash         |      189.99
(3 rows)
```

15. WHERE clause with AND

This query will return data based on a filter that deals with multiple conditions using the AND operator. The specific script goes as follows:

SELECT * FROM ORDER_INFO

WHERE CUSTOMER_ID='2' **AND** ORDER_STATUS='Delivered';

Shown in the image below is the results of this query.

WHERE Clause With BETWEEN Operator Result

```
[phfy=# SELECT * FROM ORDER_INFO
WHERE CUSTOMER_ID='2' AND ORDER_STATUS='Delivered';
 order_id | customer_id | order_date | order_status | ship_date
----------+-------------+------------+--------------+-----------
 1312     | 2           | 2020-03-10 | Delivered    | 2020-03-17
(1 row)
```

16. WHERE clause with OR

This query will return data based on a filter that deals with multiple conditions using the OR operator. The specific script goes as follows:

SELECT * FROM PAYMENT

WHERE PAYMENT_TYPE='Cash' **OR** TOTAL_PRICE='79.99';

Shown in the image below is the results of this query.

WHERE Clause With OR Result

```
phfy=# SELECT * FROM PAYMENT
WHERE PAYMENT_TYPE='Cash' OR TOTAL_PRICE='79.99';
 payment_id | order_id | customer_id | payment_type | total_price

------------+----------+-------------+--------------+------------
-
          3 | 1319     | 1           | Cash         |       79.99
          4 | 1590     | 4           | Credit *4101 |       79.99
          5 | 1368     | 2           | Cash         |      189.99
(3 rows)
```

Normalization Process:

The process of normalization, as stated by Ramez Elmasri in the textbook Fundamentals of Database Systems, "takes a relation schema through a series of tests to *certify* whether it satisfies a certain normal form." (Elmasri 2017). The process of normalization takes place in a top-down fashion, decomposing relations as necessary. Due to this fact, this process of normalization may also be recognized as relational design by analysis. Ramez Elmasri goes on to state that this process consists of five unique normal forms, all proposed based on different concepts. In most simple terms, the process of normalization is used to organize and separate a database into separate tables and columns. In this process, as the tables satisfy the database normalization form, they become less likely to conform to database modification anomalies, becoming more focused on a particular topic or purpose.

In the PHFY Database, there is lots of information being recorded. Specifically, this database consists of data regarding the customers who order products from the company, items that have been ordered, payment details, and other order information . The first step to the process of normalization in this database, is to get the data to the First Normal Form (1NF). There are several questions in which we must answer in order to do this. We must look at each table, answering whether or not the combination of all columns makes a unique row every single time, as well as determining what field can be used to uniquely identify the row. The answer to the first question is no, because there could be the same combination of data, and it would represent a different row. There also could be the same values for this row and it would be a separate row. The answer to the second question takes a little more thought. The field that can be used to uniquely identify a row in this table, is the CUSTOMER_ID. The reason for this is that it will be a different, ascending number for each customer entry that is recorded into this database. Now that we have answered these questions, our table is now in First Normal Form.

The next step in this process of normalization is to get that data into the Second Normal Form (2NF). The Second Normal Form is based on the concept of full functional dependency. According to Ramez Elmasri, "A functional dependency $X \rightarrow Y$ is a full functional dependency if removal of any attribute A from X means that the dependency does not hold anymore; that is, for any attribute $A \varepsilon X$, $(X - \{A\})$ does not functionally determine Y." (Elmasri 2017). Functional dependency means that every field that is not the primary key is determined by that primary key, so it is specific to that record. In other words, the purpose of the Second Normal Form is to Fulfill the requirements of the first normal form, while ensuring that each non-key attribute must be functionally dependent on the primary key. To do this, we must determine whether or not the columns are dependent on and specific to the primary key. In the PHFY Database, the primary key is CUSTOMER_ID, which represents the customer. Let's look at each column:

- ORDER_ID: Yes, this is dependent on the primary key because a different CUSTOMER_ID means a different order ID..
- FIRST_NAME: Yes, this is dependent on the primary key because each first name is unique to the CUSTOMER_ID.
- LAST_NAME: Yes, this is dependent on the primary key because each last name is unique to the CUSTOMER_ID.
- PHONE: Yes, this is dependent on the primary key because each phone number is unique to the CUSTOMER_ID.

- EMAIL: Yes, this is dependent on the primary key for the same reason as PHONE.
- ADDRESS: Yes, this is dependent on the primary key for the same reason as PHONE.
- ORDER_DATE: No, this is not dependent on the primary key because there may be more than one order with the same order date.
- ORDER_STATUS: No, this is not dependent on the primary key because there may be more than one order with the same delivery status.
- SHIP_DATE: No, this is not dependent on the primary key because there may be more than one order with the same shipment date.
- ITEM_ID: Yes, this is dependent on the primary key because a different CUSTOMER_ID means a different ITEM_ID for every order.
- QUANTITY: No, this is not dependent on the primary key because there may be more than one order with the same quantity of items ordered.
- LIST_PRICE: No, this is not dependent on the primary key because there may be more than one order with the same list price of the item.
- DISCOUNT: No, this is not dependent on the primary key because there may be more than one order with the same discount applied.
- PAYMENT_ID: Yes, this is dependent on the primary key because each PAYMENT_ID is unique to the CUSTOMER_ID.
- PAYMENT_TYPE: No, this is not dependent on the primary key because there may be more than one order with the same type of payment.
- TOTAL_PRICE: No, this is not dependent on the primary key because there may be more than one order with the same total price.

After going through each column, we can tell that some are dependent on the primary key, CUSTOMER_ID, while others are not. The solution to the columns that are not dependent on CUSTOMER_ID, is to simply remove them from the table, and create a new table in which they are dependent on the primary key. For the column ORDER_DATE, since it is not dependent on CUSTOMER_ID, we must remove it from the table and create a new table with ORDER_DATE within it. It will then look something like this: ORDER_INFO(ORDER_DATE). Although we have created a new table with ORDER_DATE inside of it, it is still not unique due to the fact that there could be two orders with the same ORDER_DATE. In order to fix this, we will simply create a new primary key (ORDER_ID) to the newly created ORDER_INFO table. Now the table looks like this: ORDER_INFO(<u>ORDER_ID</u>, ORDER_DATE). Next, we can add ORDER_STATUS and SHIP_DATE to the new table, resulting in the ORDER_INFO table looking like this: ORDER_INFO(<u>ORDER_ID</u>, ORDER_DATE, ORDER_STATUS, SHIP_DATE). The next step is to do the same process to the next column. The next column that we determined was not unique to the CUSTOMER_ID, was QUANTITY. We can simply remove this column from the original table, creating a new table called ORDER_ITEMS. Also, we will add a primary key named “ITEM_ID”. The new table will look like so: ORDER_ITEMS(<u>ITEM_ID</u>, QUANTITY). Next, we can simply add the next two columns to our newly created ORDER_ITEMS table, with the result of this: ORDER_ITEMS(<u>ITEM_ID</u>, QUANTITY, LIST_PRICE, DISCOUNT). The next step is to do the same process to the next column. The next column that we determined was not unique to the CUSTOMER_ID, was PAYMENT_TYPE. We can simply remove this column from the original table, creating a new table called PAYMENT. Also, we will add a primary key named "PAYMENT_ID". The new table will look like so: PAYMENT(PAYMENT_ID, PAYMENT_TYPE). Next, we can simply add the final two columns to our newly created PAYMENT table, with the result of this: PAYMENT(PAYMENT_ID, PAYMENT_TYPE, TOTAL_PRICE). We have now created three more tables to our database, with a total of four tables which go as follows:

CUSTOMER

CUSTOMER_ID	FIRST_NAME	LAST_NAME	PHONE	EMAIL	ADDRESS

ORDER_INFO

ORDER_ID	ORDER_DATE	ORDER_STATUS	SHIP_DATE

ORDER_ITEMS

ITEM_ID	QUANTITY	LIST_PRICE	DISCOUNT

PAYMENT

PAYMENT_ID	PAYMENT_TYPE	TOTAL_PRICE

Now, to link all of this data together, we will need to focus on creating foreign keys. A foreign key is a column in one table that refers to the primary key in a seperate table. Also, it is used to link one record to another based on its' unique identifier, without having to store the additional information about the linked record. We must place the primary key from one table into the other table. We will first look at the tables CUSTOMER and ORDER_INFO. Due to the fact that in this scenario, a customer can have many orders, we will place the ORDER_ID into the CUSTOMER table. Now, the CUSTOMER table will look like this, with the new foreign key being italicized: CUSTOMER(<u>CUSTOMER_ID</u>, *ORDER_ID*, FIRST_NAME, LAST_NAME, PHONE, EMAIL, ADDRESS). Next, we will look at the relationship between ORDER_INFO and ORDER_ITEMS. In this scenario, order information can have many order items, so we will place the primary key, ORDER_ID, from ORDER_INFO, into the ORDER_ITEMS table as a foreign key. Also, since order information is dependent on customers, we can place the CUSTOMER_ID inside of the table as well. Our final relationship we must focus on is between ORDER_INFO and PAYMENT. In this scenario, payments can be made for many items, so we must place the ORDER_ID into the PAYMENT table. Also, since payments are dependent on customers, we can place the CUSTOMER_ID inside of the table as well. It will look as so: PAYMENT(<u>PAYMENT_ID</u>, *ORDER_ID*, *CUSTOMER_ID*, PAYMENT_TYPE, TOTAL_PRICE). Our final set of tables go as follows, with the foreign key's being italicized:

CUSTOMER

<u>CUSTOMER_ID</u>	*ORDER_ID*	FIRST_NAME	LAST_NAME	PHONE	EMAIL	ADDRESS

ORDER_INFO

<u>ORDER_ID</u>	*CUSTOMER_ID*	ORDER_DATE	ORDER_STATUS	SHIP_DATE

ORDER_ITEMS

<u>ITEM_ID</u>	*ORDER_ID*	QUANTITY	LIST_PRICE	DISCOUNT

PAYMENT

<u>PAYMENT_ID</u>	*ORDER_ID*	*CUSTOMER_ID*	PAYMENT_TYPE	TOTAL_PRICE

Now that we have completed the Second Normal Form, we will focus on the last stage of the process of normalization.

The Third Normal Form, as stated by Ramez Elmasri, is "is based on the concept of *transitive dependency*. A functional dependency $X \rightarrow Y$ in a relation schema R is a transitive dependency if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R,11 and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold." (Elmasri 2017). In other words, the main purpose of the Third Normal Form is to fulfill the requirements of theSecond Normal Form, as well as to have no transitive functional dependency. Every non primary key attribute must depend on the primary key and nothing else. First, we will look at the CUSTOMER table. The table for CUSTOMER looks like this: CUSTOMER(CUSTOMER_ID, *ORDER_ID*, FIRST_NAME, LAST_NAME, PHONE, EMAIL, ADDRESS).. As we can see for this table, there are no columns that aren't dependent on the primary key. The Next table, ORDER_INFO, goes as follows: ORDER_INFO(ORDER_ID, CUSTOMER_ID, ORDER_DATE, ORDER_STATUS, SHIP_DATE). Again, we can determine that all of the columns in this table depend on the primary key. The last two tables, ORDER_ITEMSand PAYMENT, meet this requirement as well, maintaining that all columns depend on the primary key of their tables. Now that we have completed the process of normalization, we can see that we have developed a solid set of relationship rules, greatly improving the data structure from having almost no normalization at all. Some major benefits of the process of normalization in which we have achieved for the database include greater overall efficiency, the prevention of insert anomaly, update anomaly, delete anomaly, and the implementation of accurate data. These are just a few of the many benefits that come with the process of normalization.

Next Steps:

Every database requires a detailed plan for the next steps, improvements, modifications, and other future enhancements. As far as the PHFY Database, there are definitely many improvements and modifications that need to be made. Due to the fact that the company I have created this project for is a small start up company, I believe this current database will be a great way for the company to begin recording large pieces of data regarding their customers and order items. While I do believe that this could be a great start for the company, many steps will need to be taken in order to keep up with the company's demands. First off, I believe one major improvement to this database could be easier and faster data retrieval. This could be done through creating several indexes to provide random lookups and access to orderly records. Another future modification to the database is to just grow the amount of data that can be taken in. I believe that as every database grows, there are always more tables, fields, records and more, that are being added into the database systems. This is the case for this database, as there will likely always be new types of different data that can be added to the database. A third enhancement of this database could be employing some users and actors into the database. As mentioned earlier, this company will hire many different types of users, administrators and actors. Another possible modification that I believe every database is constantly going through, is finding the best database design. While the design of this database has gone through the Process of Normalization, adding features to the database to better the overall design will only help the overall performance of this database system. The PHFY Database is a great beginner database for Pond Hockey For You. It is an excellent way to record large pieces of data regarding customers, orders and inventory. Through the design and future modifications of this database, this company will have a well designed database for many years to come.

Resources

Brumm, B. (2020, March 31). Database Normalization: A Step-By-Step-Guide With Examples. Retrieved from https://www.databasestar.com/database-normalization/

Elmasri, R., & Navathe, S. B. (2017). *Fundamentals of database systems*. Harlow, Essex: Pearson Education.