ISTE.230.600 – Introduction to Database and Data Modeling

Fall Semester – 2023

The Hat Hub: Premier Hat Fashion Destination

Group - 3

Afra Mustafa - aim7746

Jude Abdel Halim - jma7867

Kassem Darawcha - kkd5384

Muaz Osman - mao3865

Waseem Qaffaf - whq8052

Dr. Marwan Al-Tawil

November 28, 2023

Team Manifesto: Contribution Breakdown

Team Member	Contributions
Waseem Qaffaf	Database Schema Design, Writing the Python code for interface integration
Jude Abdel Halim	E-R Diagram Creation, Relationship between entities, Project Documentation Formatting
Muaz Osman	Business scenario, Writing of the SQL Queries, DDL Commands, and DML Commands
Afra Mustafa	Key Users Tasks, Normalization and Transposition, Project Compilation
Kassem Darawcha	Algebra Notations and Operations

Table of Contents

THE HAT HUB: PREMIER HAT FASHION DESTINATION

Phase I: Conceptual and Logical Design

The Hat Hub Business Scenario

We have a shop that sells hats which is called “The Hat Hub”. Our shop has a Physical store and an E-commerce store. We need to have a more efficient way of managing our data like sales, inventory stocks, and our customer data. Our shop currently is tracking the sales and the inventory stocks using papers, which with the increasing size of data/customers will become more difficult to manage as the shop grows and expands.

To efficiently handle the inventory, sales, and customer information of our hat shop, having a database is very important. So having a database can give us a more structured and organized form of storing data like Data validation and access control to make sure that only authorized persons can access and update the data within the database.

User Tasks

1. Logging Sales: The staff must have the capability to enter details of sales in the system, including necessary information like the buyer's name, date of the transaction, details of the hat sold, the number of items, and their cost.
2. Managing inventory stock: The system should keep an updated record of the hat inventory, adjusting the stock numbers instantly with each sale. It should also permit authorized staff to update inventory details like revising the available quantities and altering prices.

3. Information Handling: The database should store client details, including their name, address, email, and contact number. Employees need to have the ability to create new customer data in the system and modify existing customer profiles as required.
4. Sales reports: The system is designed to produce reports on sales data, like the sum of monthly sales, best-selling hat models, and profit taken per customer.

List of Entities:

- Customers
- Bills
- Orders
- Delivery
- Hats

Entities Attributes:

- Customers (customer_id, name, DOB, email, contact_info, address)
- Bills (bill_id, *order_id*, tax, price, payment_method)
- Orders (order_id, *customer_id*, *hat_id*, date, quantity)
- Delivery (delivery_id, *order_id*, arrival_date)
- Hats (hat_id, brand_id, price, brand_name, style, size, quantity)

Relationships between Entities

Customers - Orders

- Each customer can place multiple orders, but each order belongs to a single customer. This establishes a one-to-many relationship between the Customers and Orders tables.
- The customer_id in the Orders table serves as a foreign key referencing the customer_id in the Customers table.

Orders - Hats

- Each order can include multiple hats, and each hat can be part of multiple orders, establishing a many-to-many relationship between the Orders and Hats tables.
- In the Orders table, the order_id serves as a foreign key referencing the primary key of the Orders table.
- In the Hats table, the hat_id serves as a foreign key referencing the primary key of the Hats table.

Orders - Delivery

- Each order is associated with a delivery, establishing a one-to-one relationship between the Orders and Delivery tables.
- The order_id in the Delivery table serves as a foreign key referencing the order_id in the Orders table.

Orders - Bills

- Each order is associated with a bill, forming a one-to-one relationship between the Orders and Bills tables.

- The order_id in the Bills table serves as a foreign key referencing the order_id in the Orders table.

Relationships between Entities Summaries

Customers - Orders

- **Relationship Name:** Place / Placed by
- **Relationship type**: one-to-many
- **Degree:** Binary
- **Description:** A customer places one or more orders, and an order is placed by one customer.
- **Minimum Cardinality: One customer can place zero or more orders (0..N).**
- **Maximum Cardinality:** There is no strict maximum limit to the number of orders a customer can place, but an order is placed by only one customer.

Orders - Hats

- **Relationship Name:** Linked / Linked by
- **Relationship type**: many-to-many
- **Degree:** Binary
- **Description:** An order is linked to one or more hats, and a hat is linked to one or more orders.
- **Minimum Cardinality:** One order can include zero or more hats (0..N)
- **Maximum Cardinality:** There is no strict maximum limit to the number of hats linked to an order, and a hat can be linked to multiple orders.

Orders - Delivery

- **Relationship Name:** Has
- **Relationship type**: one-to-one

- **Degree:** Binary
- **Description:** An order has one delivery, and a delivery is associated with one order.
- **Minimum Cardinality:** One order has zero or one delivery (0..1).
- **Maximum Cardinality:** Each order has one associated delivery, and each delivery is associated with one order.

Orders - Bills

- **Relationship Name:** Generates / Generated by
- **Relationship type**: one-to-one
- **Degree:** Binary
- **Description:** An order generates one bill, and a bill is generated by one order.
- **Minimum Cardinality:** One order generates one bill (1..1).
- **Maximum Cardinality:** Each order generates one bill, and each bill is generated by one order.

E-R diagram

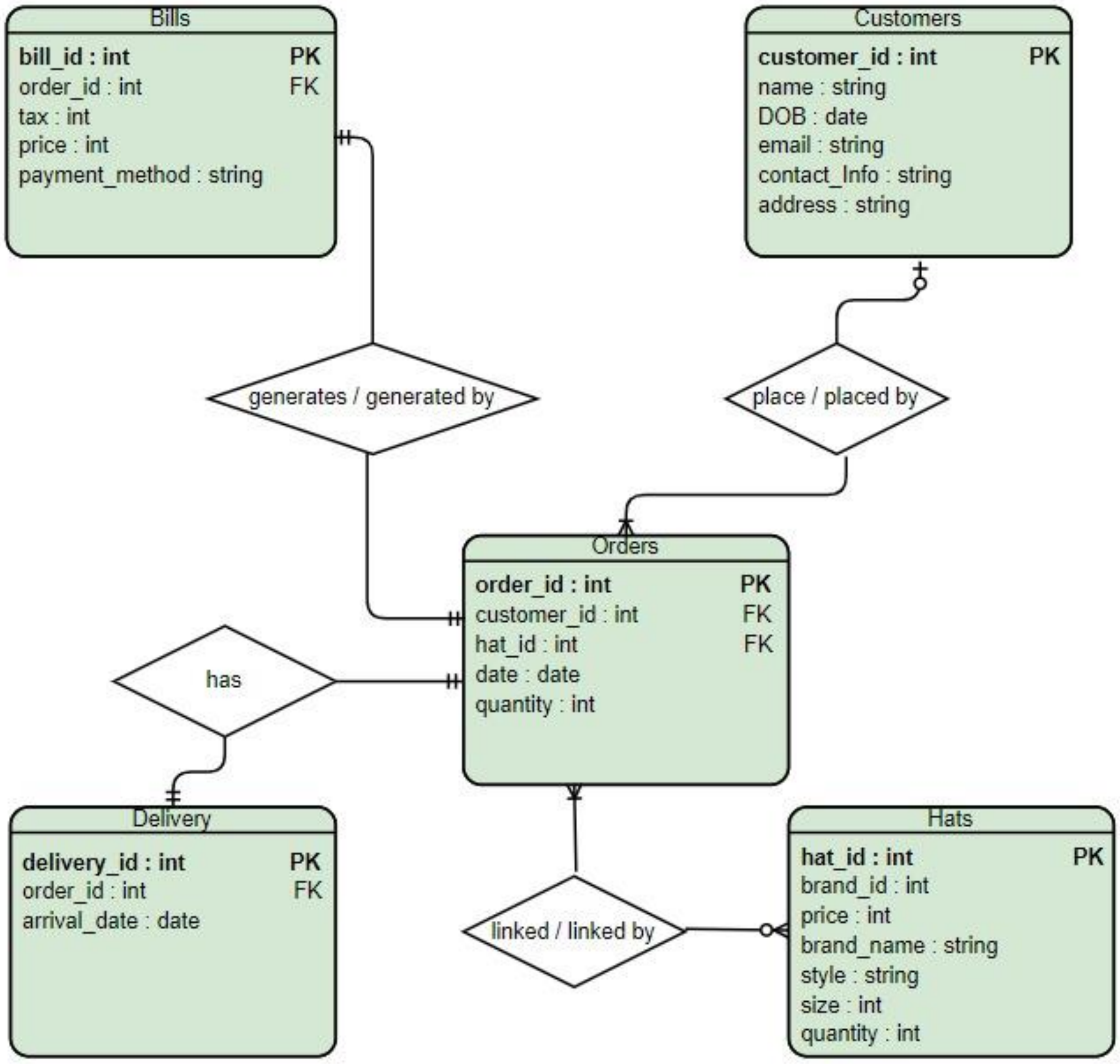

Figure 1. *E-R Diagram*

Transposition and Normalization:

In this section, we will simplify and organize the relationships in the E-R Diagram. We will use transposition and normalization to make the database more efficient and prevent any data duplication or confusion. This involves restructuring the design to make it clearer and more straightforward, ensuring an organized database.

Customers (<u>customer_id</u>, name, DOB, email, contact_info, address)

Hats (<u>hat_id</u>, brand_id, brand_name, style, size, quantity)

Orders (<u>order_id</u>, *customer_id*, *hat_id*, date, quantity)

customer_id **MEI Customers(customer_id)**

hat_id **MEI Hats(hat_id)**

Bills (<u>bill_id</u>, *order_id*, tax, price, payment_method)

order_id **MEI Orders(order_id)**

Delivery (<u>delivery_id</u>, *order_id*, arrival_date)

order_id **MEI Orders(order_id)**

First Normal Form (1st NF)

Various attributes such as size, style, color, and quantity can vary among hats, constituting repeating groups, so in this case, we will establish a distinct table named "hats_stocks," with the primary key formed by hat_id (foreign key), color, style, and size. In the orders table, a single order can include multiple hats, each with its corresponding quantity which indicates the number purchased by the customer, forming repeating groups. Consequently, a new table called "order_details" will be created, with the primary key composed of order_id (foreign key) and hat_id (foreign key). This table will also capture the quantity of each hat within an order.

Hats (hat_id, brand_id, price, brand_name)

Hats_stocks (*hat_id, color, size,style*, quantity)

***hat_id* MEI Hats(hat_id)**

Customers (customer_id , name, DOB, email, contact_info, address)

Orders (order_id, *customer_id*, date, quantity)

***customer_id* MEI Customers(customer_id)**

Order_Details (*order_id , hat_id*, quantity)

***hat_id* MEI Hats(hat_id)**

***order_id* MEI Orders(order_id)**

Bills (bill_id, *order_id*, tax, price, payment_method)

***order_id* MEI Orders(order_id)**

Delivery (delivery_id, *order_id*, arrival_date)

***order_id* MEI Orders(order_id)**

Second Normal Form (2nd NF)

The tables are already in second normal form because we don't have any partial dependencies.

Hats (hat_id, brand_id, price, brand_name)

Hats_stocks (*hat_id, color, size,style*, quantity)

***hat_id* MEI Hats(hat_id)**

Customers (<u>customer_id</u> , name, DOB, email, contact_info, address)

Orders (<u>order_id</u>, *customer_id*, date, quantity)

***customer_id* MEI Customers(customer_id)**

Order_Details (<u>*order_id* , *hat_id*</u>, quantity)

***hat_id* MEI Hats(hat_id)**

***order_id* MEI Orders(order_id)**

Bills (<u>bill_id</u>, *order_id*, tax, price, payment_method)

***order_id* MEI Orders(order_id)**

Delivery (<u>delivery_id</u>, *order_id*, arrival_date)

***order_id* MEI Orders(order_id)**

Third Normal Form (3rd NF)

In the third normal form, it's necessary to eliminate transitive dependencies so in this case, the hats table contains "brand_name," which depends on "brand_id" (a non-key attribute) rather than "hat_id." In order to fix this, we will create a new table named "brands" and relocate the relevant attributes, keeping only "brand_id" as a foreign key.

Brands (<u>brand_id</u>, brand_name)

Hats (<u>hat_id</u>, *brand_id*, price)

***brand_id* MEI Brands(brand_id)**

Hats_stocks (*hat_id, color, size,style*, quantity)

***hat_id* MEI Hats(hat_id)**

Customers (customer_id , name, DOB, email, contact_info, address)

Orders (order_id, *customer_id*, date, quantity)

***customer_id* MEI Customers(customer_id)**

Order_Details (*order_id , hat_id*, quantity)

***hat_id* MEI Hats(hat_id)**

***order_id* MEI Orders(order_id)**

Bills (bill_id, *order_id*, tax, price, payment_method)

***order_id* MEI Orders(order_id)**

Delivery (delivery_id, *order_id*, arrival_date)

***order_id* MEI Orders(order_id)**

Phase II: Physical Design:

DDL Commands

```
-- Creating the table for Customers

CREATE TABLE Customers (
    customer_id INT PRIMARY KEY NOT NULL,

    name VARCHAR(255) NOT NULL,

    DOB DATE NOT NULL,

    email VARCHAR(255) NOT NULL,

    contact_info VARCHAR(255),

    address VARCHAR(255) NOT NULL

);

-- Creating the table for Hats first, because it is referenced by other tables

CREATE TABLE Hats (

    hat_id INT PRIMARY KEY NOT NULL,

    brand_id INT NOT NULL,

    price DECIMAL(10,2) NOT NULL,

    brand_name VARCHAR(255) NOT NULL,

    style VARCHAR(255) NOT NULL,

    size INT NOT NULL,

    quantity INT NOT NULL
```

```
);

-- Creating the table for Orders after Hats table has been created

CREATE TABLE Orders (

   order_id INT PRIMARY KEY NOT NULL,

   customer_id INT NOT NULL,

   hat_id INT NOT NULL,

   date DATE NOT NULL,

   quantity INT NOT NULL,

   FOREIGN KEY (customer_id) REFERENCES Customers(customer_id),

   FOREIGN KEY (hat_id) REFERENCES Hats(hat_id)

);

-- Creating the table for Delivery

CREATE TABLE Delivery (

   delivery_id INT PRIMARY KEY NOT NULL,

   order_id INT NOT NULL,

   arrival_date DATE,
    FOREIGN KEY (order_id) REFERENCES Orders(order_id)
);
-- Creating the table for Bills
CREATE TABLE Bills (
    bill_id INT PRIMARY KEY NOT NULL,
    order_id INT NOT NULL,
    tax DECIMAL(10,2) NOT NULL,
    price DECIMAL(10,2) NOT NULL,
    payment_method VARCHAR(255) NOT NULL,
    FOREIGN KEY (order_id) REFERENCES Orders(order_id)
);
-- Creating an associative table for the many-to-many relationship between Orders and Hats
CREATE TABLE Order_Details (
    order_id INT NOT NULL,
    hat_id INT NOT NULL,
    quantity INT NOT NULL,

    PRIMARY KEY (order_id, hat_id),

    FOREIGN KEY (order_id) REFERENCES Orders(order_id),

    FOREIGN KEY (hat_id) REFERENCES Hats(hat_id)

);
```

SQL DML Sample Records

```
-- Inserting sample records into Customers

INSERT INTO Customers (customer_id, name, DOB, email, contact_info, address)
VALUES
(1, 'Kassem', '1985-04-12', 'kassem@email.com', '555-1234', '123 Street'),
(2, 'Waseem', '1990-08-25', 'waseem@email.com', '555-5678', '456 Oak Avenue'),
(3, 'Muaz', '1995-02-06', 'muaz@email.com', '555-6589', '789 Pine Road'),
(4, 'Afra', '1970-01-03', 'Afra@email.com', '555-4853', '658 Cashew Road'),
(5, 'Jude', '1975-02-03', Jude@email.com', '555-9854', '452 Almond Road');

-- Inserting sample records into Hats

INSERT INTO Hats (hat_id, brand_id, price, brand_name, style, size, quantity)
VALUES
(1, 101, 19.99, 'Nike', 'fedora', 7, 10),
(2, 102, 29.99, 'Adidas', 'beanie', 6, 15),
(3, 103, 24.99, 'PUMA', 'baseball', 7, 20);

-- Inserting sample records into Orders

INSERT INTO Orders (order_id, customer_id, hat_id, date, quantity) VALUES
(1, 1, 1, '2023-01-15', 2),
(2, 2, 2, '2023-02-20', 1),
(3, 3, 3, '2023-03-10', 3);

-- Inserting sample records into Delivery

INSERT INTO Delivery (delivery_id, order_id, arrival_date) VALUES
(1, 1, '2023-01-20'),
(2, 2, '2023-02-25');

-- Inserting sample records into Bills

INSERT INTO Bills (bill_id, order_id, tax, price, payment_method) VALUES
(1, 1, 1.50, 39.98, 'credit card'),
(2, 2, 2.25, 29.99, 'credit card'),
(3, 3, 1.80, 74.97, 'debit card');

-- Inserting sample records into Order_Details for many-to-many
relationship including quantity

INSERT INTO Order_Details (order_id, hat_id, quantity) VALUES
(1, 1, 2), -- Assuming order 1 includes 2 of hat 1
```

(2, 2, 1), -- Assuming order 2 includes 1 of hat 2

(3, 3, 3), -- Assuming order 3 includes 3 of hat 3

(3, 1, 1); -- Assuming order 3 includes 1 of hat 1, this order includes multiple types of hats

SQL Queries

Query 1: DELETE using IS NULL

- This query would delete records from the Hats table where the Quantity is NULL.
- DELETE FROM Hats WHERE quantity IS NULL; -- Assuming no hats should have a null quantity, but the condition ensures no actual deletion.

Query 2: SELECT with INNER JOIN and LIKE

- This query retrieves all records that join the **Hats** and **Order_Details** tables based on a condition that involves a text pattern (LIKE).
- SELECT * FROM Hats INNER JOIN Order_Details ON Hats.hat_id = Order_Details.hat_id WHERE Hats.style LIKE 'fedo%';

Query 3: SELECT with UNION ALL

- This query selects all records from both **Hats** and **Orders** tables. It combines all records, including duplicates.
- SELECT * FROM Hats UNION ALL SELECT * FROM Orders;

Query 4: SELECT with EXISTS and subquery

- This query selects the **brand_name** from **Hats** where there exists a corresponding entry in the **Order_Details** table with a **quantity** less than 5.
- SELECT brand_name FROM Hats WHERE EXISTS (SELECT * FROM Order_Details WHERE Hats.hat_id = Order_Details.hat_id AND quantity < 5);

Query 5: Retrieve all customers and their orders:

```
SELECT Customers.name, Orders.order_id, Hats.brand_name, Hats.style, Orders.quantity
FROM Customers JOIN Orders ON Customers.customer_id = Orders.customer_id JOIN Hats
ON Orders.hat_id = Hats.hat_id;
```

Algebra Notations

Example Query:

Retrieve a list of all bills with the customer's name, bill ID, and total price for orders placed by customers who have purchased hats from the brand named "Hat Hub”.

a. Projection, Selection, and Joins

Relational Algebra:

π name, bill_id, (tax + price) AS total_price

(Orders ⋈ Customers ⋈ Hats) ⋈ Bills

σ brand_name = 'HatHub'

SQL

```
SELECT Customers.name, Bills.bill_id, (Bills.tax + Bills.price) AS total_price
FROM Orders
JOIN Customers ON Orders.customer_id = Customers.customer_id
JOIN Hats ON Orders.hat_id = Hats.hat_id
JOIN Bills ON Orders.order_id = Bills.order_id
WHERE Hats.brand_name = 'HatHub';
```

Explanation

Projection (π): The query selects specific columns (name, bill_id, and (tax + price) AS total_price) from the combined Orders, Customers, Hats, and Bills tables. The total price is calculated by adding the tax and price columns from the Bills table.

Join (⋈): Joins are performed between Orders, Customers, Hats, and Bills tables based on their foreign key relationships (customer_id, hat_id, and order_id). This creates a unified table with information about customers, orders, hats, and bills.

Selection (σ): A selection operation is applied to filter rows where the hat's brand name is 'HatHub.' This ensures that only orders for hats from the specified brand are considered.

b. Union, Intersection, and Selection:

Relational Algebra

σ color = 'green' ((Sneakers ⋈ Sneaker_inventory) ⋂ (Sneakers ⋈ σ color = 'green' (Sneaker_inventory)))

SQL

```
SELECT size
FROM Sneakers
JOIN Sneaker_inventory ON Sneakers.sneaker_id = Sneaker_inventory.sneaker_id
WHERE color = 'green'
INTERSECT
SELECT size
FROM Sneakers
JOIN Sneaker_inventory ON Sneakers.sneaker_id = Sneaker_inventory.sneaker_id
WHERE color = 'green';
```

Explanation

Join (⋈): Connects the Sneakers and Sneaker_inventory tables based on the sneaker_id column.

Selection (σ): Filters rows where the color is 'green' from the combined Sneakers and Sneaker_inventory tables.

Intersection (⋂): Performs a set intersection on the combined table to select only the sizes that are available in both tables for the specified color.

c. Projection, Union, and Joins

Relational Algebra

π name, bill_id, (tax + price) AS total_price

(Orders ⋈ Customers ⋈ Hats) ⋈ Bills

∪

π name, bill_id, (tax + price) AS total_price

(Orders ⋈ Customers ⋈ Hats) ⋈ Bills

SQL

```
SELECT Customers.name, Bills.bill_id, (Bills.tax + Bills.price) AS total_price
FROM Orders
JOIN Customers ON Orders.customer_id = Customers.customer_id
JOIN Hats ON Orders.hat_id = Hats.hat_id
JOIN Bills ON Orders.order_id = Bills.order_id
```

UNION

```
SELECT Customers.name, Bills.bill_id, (Bills.tax + Bills.price) AS total_price
FROM Orders
JOIN Customers ON Orders.customer_id = Customers.customer_id
JOIN Hats ON Orders.hat_id = Hats.hat_id
JOIN Bills ON Orders.order_id = Bills.order_id;
```

Explanation

Projection (π): Selects specific columns (name, bill_id, and (tax + price) AS total_price) from the combined Orders, Customers, Hats, and Bills tables. The total price is calculated by adding the tax and price columns from the Bills table.

Join (⋈): Joins are performed between Orders, Customers, Hats, and Bills tables based on their foreign key relationships (customer_id, hat_id, and order_id). This creates a unified table with information about customers, orders, hats, and bills.

Union (∪): Combines the results of two SELECT statements, eliminating duplicate rows. The UNION operator is used to merge the information about bills for all customers from the two sets of joined tables.

In conclusion, This query retrieves a list of all bills with the customer's name, bill ID, and total price for orders placed by customers who have purchased hats from the brand named "HatHub." The explanation breaks down the query into its relational algebra components and explains the corresponding SQL operations.