# Jitters: Use Case Study Report

**By: Atera Alam and Priska Mohunsingh**

## I. Executive Summary

The purpose of this project was to create a database system for Jitters, an expanding coffee shop that needed to run their business seamlessly. This project specifically focused on the customer ordering process and the inventory ordering process. Jitters, which has been rapidly growing in popularity, struggled in their day-to-day operations with increasing customer demand and a lack of an organized database system in place. By implementing this database system, the crucial information needed to run the operations of the business and utilize the information of existing customers will become simple.

To tackle the specific business needs of Jitters, a foundational database structure was first created. After creating the database structure, a conceptual model was built using the information from the preliminary database structure. Then, after creating the conceptual model by drawing Enhanced Entity Relation (EER) and Unified Modeling Language (UML) diagrams, a relational model was authored. Then, using the relational model, a database was created using MySQL. MySQL is a popular RDBMS system that is easily accessible and user-friendly, which made it a good choice for this project. To populate the database with data, a Python script was used. After which, MongoDB was used for the NoSQL implementation of this project. MongoDB was chosen for NoSQL implementation as MongoDB is a document-based DBMS and can scale horizontally, which makes it a great option for Jitters.

This project was successful in addressing the business problem Jitters was facing. The project created a robust database for the business that tracked crucial customer information and inventory information and was designed to retrieve necessary data easily through queries. Database access through Python is helpful for businesses to draw necessary information and create visualizations using Python tools. The next step for this project can be to integrate more branches of this business into the Jitters database so that all crucial business data is collected and utilized to make crucial business decisions.

## II. Introduction

Jitters, a beloved coffee business in New York City, has joined the popular beverage industry with a unique spin on a better ordering system for customer success. As the company expanded, it faced a critical need for a centralized system to manage its growing operations effectively. The surge in popularity highlighted an imperative need for an advanced data management solution. This necessity led us to build this database project, aimed at transforming Jitters' operations through technology.

The project centered on enhancing Jitters' operational efficiency, customer service, and inventory management. By automating the ordering systems for both customers and inventory, we set out to streamline processes and improve the overall customer experience. The

introduction of a relational database was the cornerstone of this endeavor, streamlining Jitters' operations by eradicating data redundancies and enhancing the customer service experience.

The deployment of the database heralds a significant cutback in the time required to process customer orders, anticipated to be reduced by more than 50%, as it eliminates the superfluous tasks characteristic of manual operations. The implementation of a unified system for inventory control predicates that a solitary data entry can actuate the automated procurement protocols, thus maintaining ideal stock levels without the necessity for recurrent manual entries. This enhancement is further instrumental in enabling synchronous data interchange across the various outlets of Jitters, thereby integrating inventory and sales data into a unified operational strategy.

In the realm of data acquisition, the initial endeavor was to construct a comprehensive dataset that accurately mirrored the transactional dynamics of Jitters. This compilation of sample data facilitated the simulation of diverse operational contexts, establishing a foundation for a durable and adaptable system.

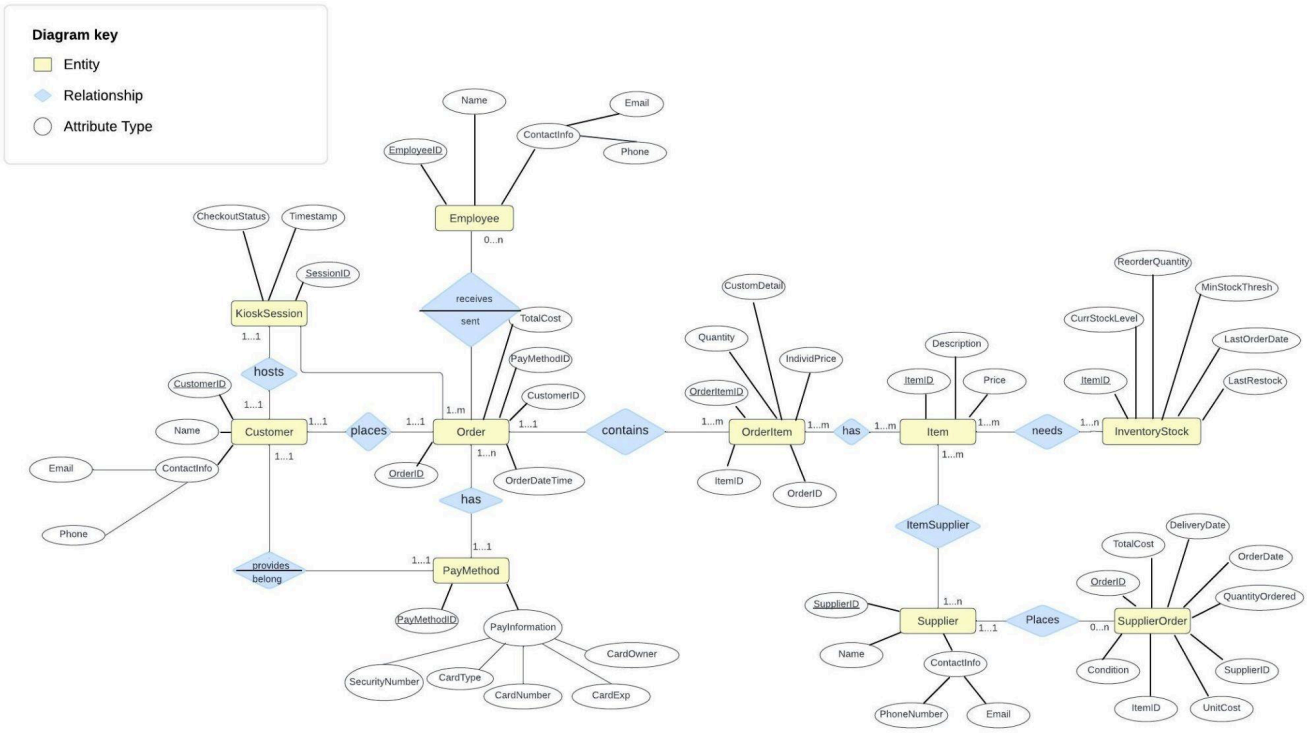
The database's architectural progression was marked by a deliberate and methodical approach, commencing with a conceptual framework that identified and mapped the interrelationships between key entities. Progressing through to a logical schema, it articulated the nuances of data classifications and limitations. The culmination of this process was the materialization of the database in a physical form, employing MySQL to anchor the relational structure, and MongoDB to manage the elements of unstructured data, thereby amalgamating SQL's dependability with the versatility afforded by NoSQL technologies.

Python and its suite of visualization libraries, such as Pandas and Matplotlib, constituted the backbone of the data strategy, empowering the analysis of complex datasets and the graphical representation of vital business indicators. This confluence of database infrastructure and analytical capabilities bestowed upon Jitters a transformative resource—a dynamic database system that not only rectified existing data quandaries but also established an extensible base for future expansion.

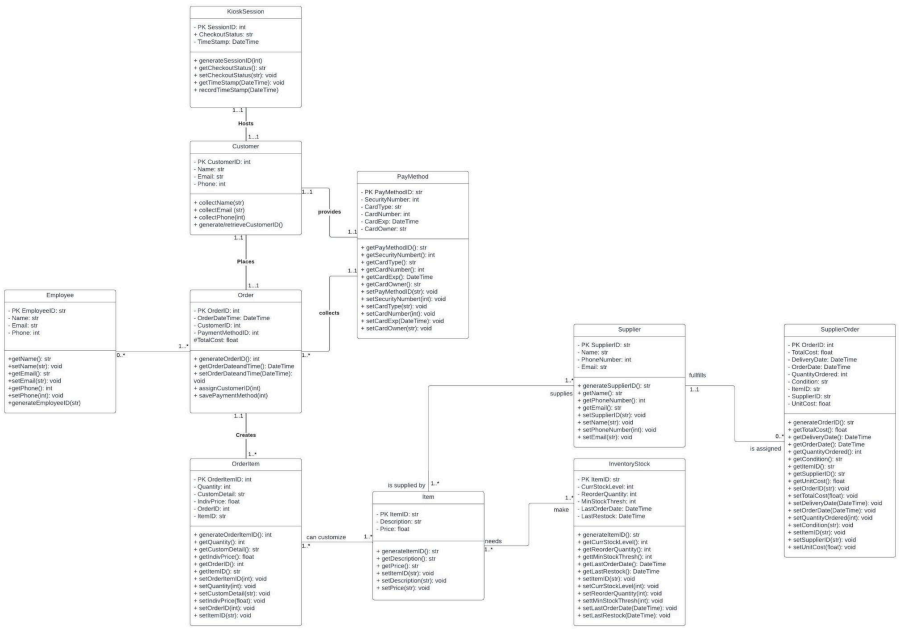
# III. Conceptual Data Modeling

Please refer to the definitions of attributes and entities shown in the diagrams in the "References" section.

## 1. EER Diagram




## 2. UML Class Diagram



## IV. Mapping Conceptual Model to Relational Model

**Primary Key (PK): underlined; Foreign Key (FK): italicized**

- KioskSession (SessionID, Timestamp, CheckoutStatus)
- Customer (CustomerID, Name)
- CustomerContactInfo (*Customer ID*, Email, Phone)
  FK - not null. CustomerID refers to customerID in customer relation
- Order (OrderID, OrderDateTime, *CustomerID*, PaymentMethodID, TotalCost)
  FK - not null. CustomerID refers to customerID in customer and customer contact info relations.
- OrderItem (OrderItemID, Quantity, CustomDetail, IndivPrice, OrderID, ItemID)
- PayMethod (PayMethodID, Security Number, Card Type, Card Number, Card Exp, Card Owner)
- Item (ItemID, Description, Price)
- Supplier (SupplierID, Name, PhoneNumber, Email)
- InventoryStock (*ItemID,* CurrStockLevel, ReorderQuantity, MinStockThresh, LastOrderDate, LastRestock)
  FK- not null. ItemID refers to the ItemID in Item relation.
- SupplierOrder (OrderID, TotalCost, DeliveryDate, OrderDate, QuantityOrdered, Condition, ItemID, SupplierID, UnitCost)
- Employee (EmployeeID, Name, Email, Phone)
- EmployeeWorksOrder (*EmployeeID, OrderID)*
  FK EmployeeID and Order ID are not null. Employee ID refers to EmployeeID in employee relation. OrderID refers to OrderID in order relation.
- ItemInStock (*ItemID*)
  FK ItemID is not null. Item ID refers to ItemID in both the Item and InventoryStock relations.
- OrderItemhasItem *(OrderItemID, IndivItemID)*
  Both FKs are not null. OrderitemID refers to the OrderItemID in OrderItem relation. IndivItemID refers to itemID in Item relation)
- ItemSupplier (*ItemID, SupplierID)*
  FK ItemID refers to ItemID in item relation. SupplierID refers to SupplierID in Supplier relation. Here, both FKs are not null.

## V. Implementation of Relational Model via MySQL and NoSQL

### MySQL Implementation

We created the database in MySQL Workbench 8.0 CE. Upon creating JittersDB, we created all of the tables mapped out in our EER Model and UML Class Diagram. We then generated random, appropriate data into each of the tables using a script we developed in Python. Lastly, we developed several different queries to ensure the database is running as expected and delivering data that can be used to further analyze.

## 1. Database Creation in MySQL

We successfully created the database (JittersDB) and 14 tables mapped out using the "CREATE TABLE" function, which is pre-built in MySQL. We had to ensure primary and foreign keys were referenced accurately, and that no table being created was duplicating another's.

```sql
CREATE DATABASE IF NOT EXISTS JittersDB;
USE JittersDB;
```

```sql
CREATE TABLE Customer (
    CustomerID INT AUTO_INCREMENT PRIMARY KEY,
    Name VARCHAR(255) NOT NULL
);
```

```sql
CREATE TABLE Supplier (
    SupplierID CHAR(20) PRIMARY KEY NOT NULL,
    Name VARCHAR(255) NOT NULL,
    PhoneNumber VARCHAR(20) UNIQUE NOT NULL,
    Email VARCHAR(255) UNIQUE NOT NULL
);
```

```sql
CREATE TABLE Item (
    ItemID INT NOT NULL PRIMARY KEY,
    Description VARCHAR(255),
    Price DOUBLE(5,2)
);
```

```sql
CREATE TABLE Employee (
    EmployeeID INT AUTO_INCREMENT PRIMARY KEY,
    Name VARCHAR(255) NOT NULL,
    Email VARCHAR(255) NOT NULL,
    Phone VARCHAR(20)
);
```

```sql
CREATE TABLE PAY_METHOD (
    PayMethodID CHAR(255) PRIMARY KEY,
    Security_Number INT(10),
    Card_Number CHAR(255),
    CardType CHAR(10) NOT NULL,
    Card_Exp VARCHAR(8) NOT NULL,
    Card_Owner VARCHAR(255) NOT NULL
);
```

```sql
ALTER TABLE PAY_METHOD ADD PRIMARY KEY (PayMethodID);

CREATE TABLE Customer_Contact_Info (
    CustomerID INT NOT NULL PRIMARY KEY,
    Email VARCHAR(255) NOT NULL,
    PhoneNumber CHAR(15) NOT NULL,
    CONSTRAINT UC1 UNIQUE(Email)
);
```

```sql
CREATE TABLE OrderTable (
    OrderID INT AUTO_INCREMENT PRIMARY KEY,
    CustomerID INT NOT NULL,
    OrderDateTime DATETIME NOT NULL,
    PayMethodID CHAR(255) NOT NULL,
    TotalCost DECIMAL (10,2) NOT NULL,
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID),
    FOREIGN KEY (PayMethodID) REFERENCES PAY_METHOD(PayMethodID)
);
```

```sql
CREATE TABLE InventoryStock (
    ItemID INT NOT NULL,
    CurrStockLevel INT NOT NULL,
    ReorderQuantity INT NOT NULL,
    MinStockThresh INT NOT NULL,
    LastOrderDate DATETIME,
    LastRestock DATETIME,
    PRIMARY KEY (ItemID),
    FOREIGN KEY (ItemID) REFERENCES Item(ItemID)
);
```

```sql
CREATE TABLE ORDERITEM (
    OrderItemID INT NOT NULL PRIMARY KEY,
    Quantity INT NOT NULL,
    CustomDetail VARCHAR(255),
    IndivPrice DOUBLE(5,2),
    OrderID INT NOT NULL,
    ItemID INT NOT NULL,
    FOREIGN KEY (OrderID) REFERENCES OrderTable(OrderID),
    FOREIGN KEY (ItemID) REFERENCES Item(ItemID)
);
```

```sql
CREATE TABLE SupplierOrder (
    OrderID INT AUTO_INCREMENT PRIMARY KEY,
    TotalCost DECIMAL(10,2) NOT NULL,
    DeliveryDate DATETIME,
    OrderDate DATETIME NOT NULL,
    QuantityOrdered INT NOT NULL,
    `Condition` VARCHAR(255),
    ItemID INT NOT NULL,
    SupplierID CHAR(20) NOT NULL, -- Adjusted to match Supplier table's SupplierID type
    UnitCost DECIMAL(10,2) NOT NULL,
    FOREIGN KEY (ItemID) REFERENCES Item(ItemID),
    FOREIGN KEY (SupplierID) REFERENCES Supplier(SupplierID)
);
```

```sql
CREATE TABLE EmployeeWorksOrder (
    EmployeeID INT NOT NULL,
    OrderID INT NOT NULL,
    FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID),
    FOREIGN KEY (OrderID) REFERENCES OrderTable(OrderID),
    PRIMARY KEY (EmployeeID, OrderID)
);

CREATE TABLE OrderItemhasItem (
    OrderItemID INT NOT NULL,
    IndivItemID INT NOT NULL,
    FOREIGN KEY (OrderItemID) REFERENCES OrderItem(OrderItemID),
    FOREIGN KEY (IndivItemID) REFERENCES Item(ItemID),
    PRIMARY KEY (OrderItemID, IndivItemID)
);
```

```sql
CREATE TABLE ItemSupplier (
    ItemID INT NOT NULL,
    SupplierID CHAR(20) NOT NULL,
    FOREIGN KEY (ItemID) REFERENCES Item(ItemID),
    FOREIGN KEY (SupplierID) REFERENCES Supplier(SupplierID),
    PRIMARY KEY (ItemID, SupplierID)
);

CREATE TABLE Kiosk_Session(
    SessionID CHAR(7) NOT NULL PRIMARY KEY,
    Timestamp DATETIME,
    CheckoutStatus BOOLEAN
);
```

## 2. Data Generation and Loading

We wrote a Python script to generate data. Please see the following code that we crafted and used. First, we defined helper functions in the script to generate random pieces of data for all strings, datetimes, emails, phone numbers, etc. We then generated data for independent tables that did not have foreign key dependencies (tables like ‘kiosk_session’). Afterwards, we progressed into dependent tables (i.e., ‘OrderTable’ depends on ‘Customer’). Afterwards, we handled the many-to-many relationships (i.e., ‘OrderItemhasItem’); creating associations between the relevant entities. Throughout the data generation process, we ensured the script respected the dependencies between tables due to the foreign keys. Finally, for each piece of generated data, the script prints out a SQL ‘INSERT’ statement. This populates our database with generated data.

Please see the following tables with data:

```sql
SELECT * FROM Kiosk_Session LIMIT 10;
```

Result Grid | Filter Rows: | Edit:

| SessionID | Timestamp | CheckoutStatus |
|---|---|---|
| A2X8J4K | 2021-05-14 18:45:30 | 1 |
| B3Y9L5L | 2022-07-22 07:22:16 | 0 |
| C4Z0M6M | 2023-02-11 12:00:00 | 1 |
| D5X1N7N | 2020-11-03 23:15:45 | 0 |
| E6W2O8O | 2021-09-19 05:30:30 | 1 |
| F7V3P9P | 2022-03-28 13:47:55 | 0 |
| G8U4Q0Q | 2023-08-06 16:25:10 | 1 |
| H9T5R1R | 2020-12-25 19:05:05 | 0 |
| I0S6S2S | 2021-07-04 21:45:20 | 1 |
| IQ7XB1W | 2020-04-23 22:33:42 | 0 |
| NULL | NULL | NULL |

```sql
SELECT * FROM OrderTable LIMIT 10;
```

Result Grid | Filter Rows: | Edit:

| OrderID | CustomerID | OrderDateTime | PayMethodID | TotalCost |
|---|---|---|---|---|
| 1 | 10 | 2022-01-19 00:26:06 | 1 | 39.47 |
| 2 | 2 | 2022-04-07 23:41:28 | 4 | 71.23 |
| 3 | 9 | 2021-11-07 19:54:36 | 5 | 354.53 |
| 4 | 9 | 2022-01-21 03:40:48 | 3 | 299.03 |
| 5 | 10 | 2020-09-05 08:03:37 | 1 | 96.30 |
| 6 | 7 | 2023-07-08 13:06:09 | 3 | 188.88 |
| 7 | 4 | 2022-04-17 17:33:33 | 1 | 370.87 |
| 8 | 4 | 2022-07-15 03:39:16 | 3 | 127.83 |
| 9 | 10 | 2022-02-11 14:48:48 | 3 | 292.29 |
| 10 | 2 | 2020-08-12 03:57:45 | 4 | 363.23 |

```sql
SELECT * FROM OrderTable LIMIT 10;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Co

| ItemID | CurrStockLevel | ReorderQuantity | MinStockThresh | LastOrderDate | LastRestock |
|---|---|---|---|---|---|
| 1 | 29 | 5 | 3 | 2021-10-14 20:42:29 | 2022-01-17 16:17:28 |
| 2 | 17 | 10 | 4 | 2021-01-04 09:29:47 | 2021-09-07 18:59:51 |
| 3 | 63 | 9 | 5 | 2023-07-09 09:41:01 | 2023-11-23 23:20:11 |
| 4 | 51 | 12 | 2 | 2021-06-25 23:04:41 | 2021-07-16 02:14:27 |
| 5 | 81 | 20 | 7 | 2022-07-01 20:34:42 | 2023-04-18 23:01:24 |
| 6 | 14 | 18 | 10 | 2020-10-30 10:06:03 | 2020-01-11 05:33:53 |
| 7 | 73 | 16 | 9 | 2023-10-12 08:15:55 | 2021-12-11 06:57:18 |
| 8 | 60 | 3 | 3 | 2021-11-15 12:21:43 | 2022-06-23 10:27:33 |
| 9 | 72 | 9 | 10 | 2023-03-27 10:37:10 | 2021-01-10 23:31:56 |
| 10 | 13 | 12 | 6 | 2021-05-25 12:49:33 | 2023-10-09 07:35:48 |

## 3. SQL Queries

***Query 1:*** This query displays customer orders along with their names and contact information.

```
SELECT
    o.OrderID,
    o.OrderDateTime,
    o.TotalCost AS OrderTotalCost,
    c.Name AS CustomerName,
    cci.Email AS CustomerEmail,
    cci.PhoneNumber AS CustomerPhoneNumber
FROM
    `OrderTable` o
INNER JOIN Customer c ON o.CustomerID = c.CustomerID
INNER JOIN Customer_Contact_Info cci ON c.CustomerID = cci.CustomerID
ORDER BY o.OrderDateTime DESC;
```

| OrderID | OrderDateTime | OrderTotalCost | CustomerName | CustomerEmail | CustomerPhoneNumber |
|---|---|---|---|---|---|
| 6 | 2023-07-08 13:06:09 | 188.88 | Customer7 | customer7.60@mail.com | 9546536000 |
| 8 | 2022-07-15 03:39:16 | 127.83 | Customer4 | customer4.9@mail.com | 9255160764 |
| 7 | 2022-04-17 17:33:33 | 370.87 | Customer4 | customer4.9@mail.com | 9255160764 |
| 2 | 2022-04-07 23:41:28 | 71.23 | Customer2 | customer2.19@example.com | 2207130431 |
| 9 | 2022-02-11 14:48:48 | 292.29 | Customer10 | customer10.100@test.org | 3846850064 |
| 4 | 2022-01-21 03:40:48 | 299.03 | Customer9 | customer9.19@mail.com | 8973348952 |
| 1 | 2022-01-19 00:26:06 | 39.47 | Customer10 | customer10.100@test.org | 3846850064 |
| 3 | 2021-11-07 19:54:36 | 354.53 | Customer9 | customer9.19@mail.com | 8973348952 |
| 5 | 2020-09-05 08:03:37 | 96.30 | Customer10 | customer10.100@test.org | 3846850064 |
| 10 | 2020-08-12 03:57:45 | 363.23 | Customer2 | customer2.19@example.com | 2207130431 |

***Query 2:*** This query uses the aggregate function group by/having. We calculated the number of orders and the total amount spent per customer, and also displayed the customer ID and name. We then grouped the results by the customer ID and name of those who have placed over 3 orders or spent over 50 dollars.

```
SELECT
    c.CustomerID,
    c.Name AS CustomerName,
    COUNT(o.OrderID) AS NumberOfOrders,
    SUM(o.TotalCost) AS TotalSpent
FROM
    `OrderTable` o INNER JOIN Customer c ON o.CustomerID = c.CustomerID
GROUP BY c.CustomerID, c.Name
HAVING COUNT(o.OrderID) > 3 OR SUM(o.TotalCost) > 50;
```

Result Grid | Filter Rows: | Export:

| CustomerID | CustomerName | NumberOfOrders | TotalSpent |
|---|---|---|---|
| 10 | Customer10 | 3 | 428.06 |
| 2 | Customer2 | 2 | 434.46 |
| 9 | Customer9 | 2 | 653.56 |
| 7 | Customer7 | 1 | 188.88 |
| 4 | Customer4 | 2 | 498.70 |

***Query 3:*** This nested query finds the names of the customers whose total order cost is higher than the average total cost.

```
SELECT
    c.CustomerID,
    c.Name AS CustomerName,
    o.OrderID,
    o.TotalCost
FROM
    `OrderTable` o
INNER JOIN Customer c ON o.CustomerID = c.CustomerID
WHERE o.TotalCost > (
    SELECT AVG(TotalCost)
    FROM `OrderTable`
)
ORDER BY o.TotalCost DESC;
```

Result Grid | Filter Rows: | Export:

| CustomerID | CustomerName | OrderID | TotalCost |
|---|---|---|---|
| 4 | Customer4 | 7 | 370.87 |
| 2 | Customer2 | 10 | 363.23 |
| 9 | Customer9 | 3 | 354.53 |
| 9 | Customer9 | 4 | 299.03 |
| 10 | Customer10 | 9 | 292.29 |

***Query 4:*** This query finds the names and the total number of employees.

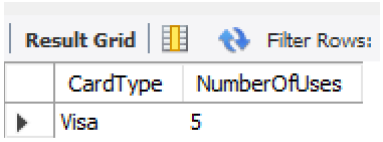
```
SELECT
      e.name,
      (SELECT COUNT(*) FROM Employee) AS TotalEmployee
FROM
       Employee e;
```

Result Grid | Filter Rows:

| name | TotalEmployee |
|---|---|
| Employee1 | 10 |
| Employee2 | 10 |
| Employee3 | 10 |
| Employee4 | 10 |
| Employee5 | 10 |
| Employee6 | 10 |
| Employee7 | 10 |
| Employee8 | 10 |
| Employee9 | 10 |
| Employee10 | 10 |

***Query 5:*** This query finds the most used card payment method among customers.

```
SELECT
    CardType,
    COUNT(*) AS NumberOfUses
FROM
    Pay_Method
GROUP BY CardType
ORDER BY NumberOfUses DESC
LIMIT 1;
```



Result Grid | Filter Rows:

| CardType | NumberOfUses |
|---|---|
| Visa | 5 |

***Query 6:*** This nested subquery lists customers and the number of their orders, but only for those customers who have at least one order with a total cost greater than $100.

```
SELECT
       Name,
       (SELECT COUNT(*) FROM OrderTable WHERE CustomerID = c.CustomerID) as
OrderCount
FROM Customer c
WHERE EXISTS (SELECT 1 FROM OrderTable WHERE CustomerID = c.CustomerID AND TotalCost
> 100);
```

| Name | OrderCount |
|---|---|
| Customer2 | 2 |
| Customer4 | 2 |
| Customer7 | 1 |
| Customer9 | 2 |
| Customer10 | 3 |

***Query 7:*** This query calculates the total amount spent by each customer grouped by their payment card type and orders the results from highest to lowest spender.

```
SELECT
       customer.Name,
       pay_method.CardType,
       SUM(ordertable.TotalCost) AS TotalSpent
FROM
       ordertable
JOIN
       customer ON ordertable.CustomerID = customer.CustomerID
JOIN
       pay_method ON ordertable.PayMethodID = pay_method.PayMethodID
GROUP BY
       customer.CustomerID, pay_method.CardType
ORDER BY
       TotalSpent DESC;
```

| Name | CardType | TotalSpent |
|---|---|---|
| Customer9 | Visa | 653.56 |
| Customer4 | Visa | 498.70 |
| Customer2 | Visa | 434.46 |
| Customer10 | Visa | 428.06 |
| Customer7 | Visa | 188.88 |

***Query 8:*** This complex query retrieves low inventory levels which can help in reordering.

```
SELECT
	Item.Description,
	InventoryStock.CurrStockLevel,
	InventoryStock.MinStockThresh,
	Supplier.Name AS LastSupplier,
	MAX(SupplierOrder.OrderDate) AS LastOrderDate
FROM
	InventoryStock
JOIN
	Item ON InventoryStock.ItemID = Item.ItemID
LEFT JOIN
	SupplierOrder ON InventoryStock.ItemID = SupplierOrder.ItemID
LEFT JOIN
	Supplier ON SupplierOrder.SupplierID = Supplier.SupplierID
WHERE
	InventoryStock.CurrStockLevel <= InventoryStock.MinStockThresh
GROUP BY
	Item.Description, InventoryStock.CurrStockLevel,
InventoryStock.MinStockThresh, LastSupplier
ORDER BY
	LastOrderDate DESC;
```

| Description | CurrStockLevel | MinStockThresh | LastSupplier | LastOrderDate |
|---|---|---|---|---|
| Matcha Latte | 5 | 10 | NULL | NULL |
| Iced Chai Latte | 10 | 30 | NULL | NULL |

## Query 9:

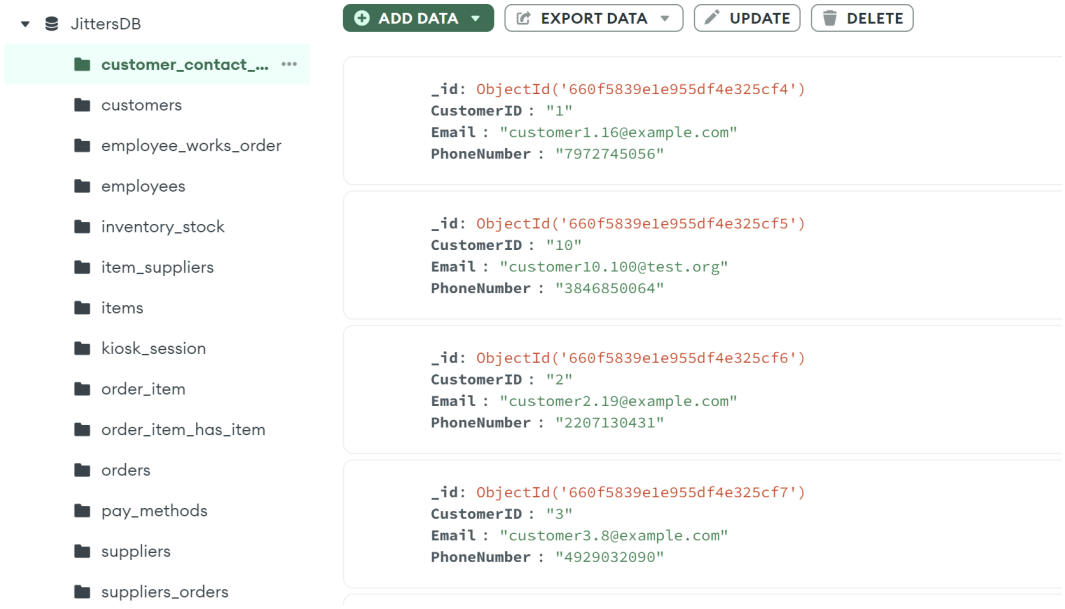
SELECT Name FROM Customer
UNION
SELECT Name FROM Employee;

This shows customers who are also employees

## NoSQL Implementation

We created JittersDB in MongoDBCompass, and then created the 14 *collections*, representing each of the tables in our EER Model. Upon creating these collections, we crafted a Python script to transfer over all of the generated data from the MySQL version of JittersDB into our MongoDBCompass version of JittersDB. This required us to connect to both mysql and mongodb setups within the python script. Please refer to the “References” section to see the code we crafted and used to transfer this data accurately.

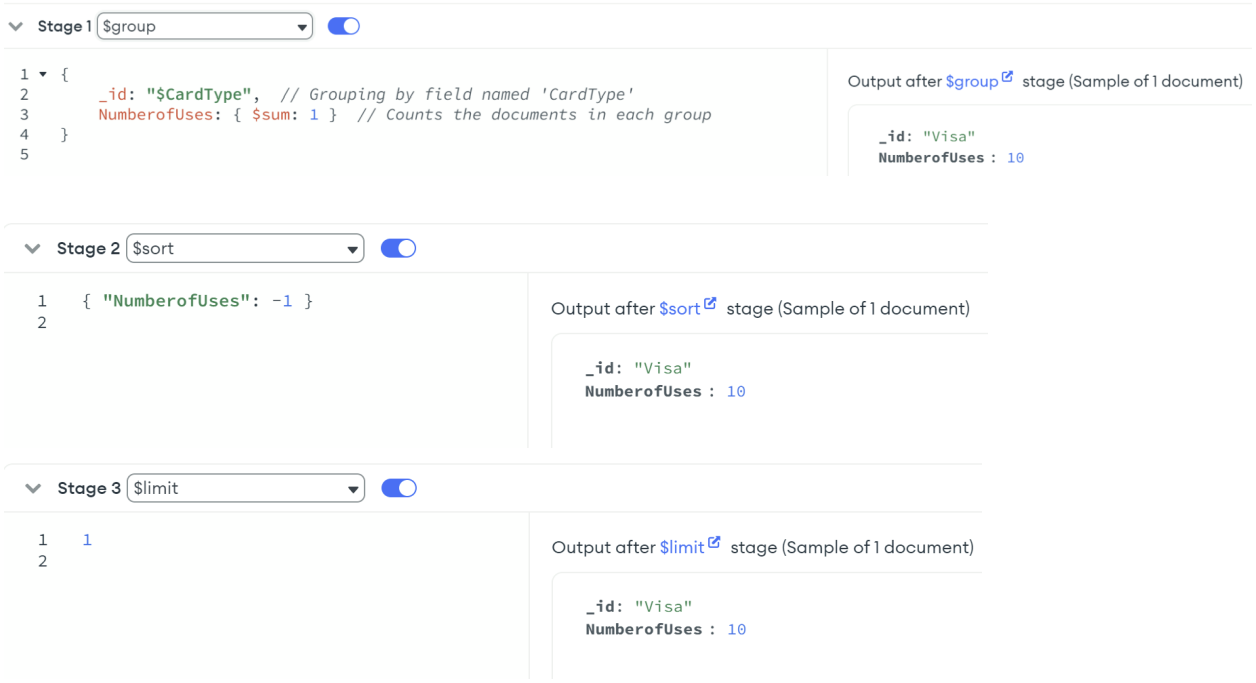
### 1. Collections Created




### 2. NoSQL Queries

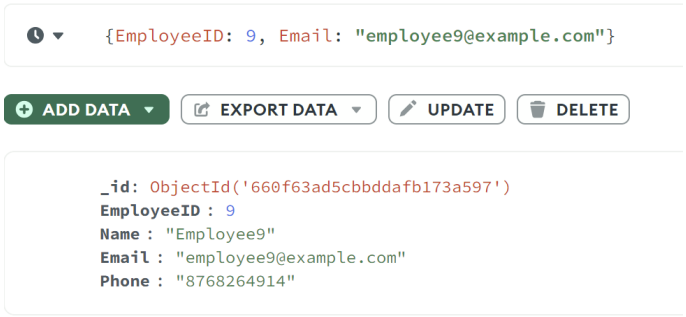
***Query 1:*** This is the MongoDB query adaptation of query 5 in our MySQL query implementation.

```
db.pay_method.aggregate([
{
  $group: {
  _id: "$CardType",
  NumberofUses: {$sum: 1}
  }
},
  {$sort: NumberofUses: -1}
},
{
  {$limit: 1}
}])
```



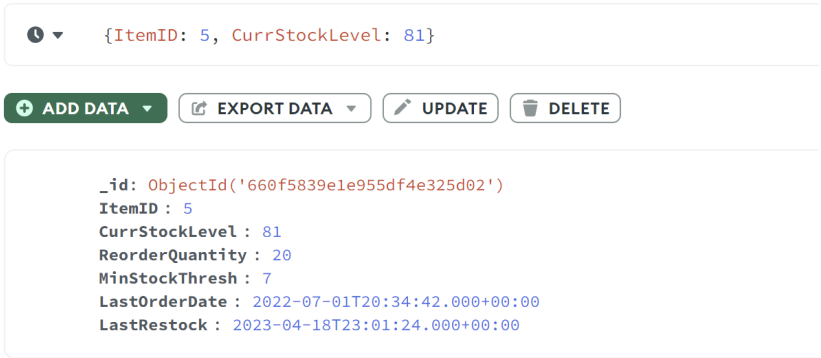

***Query 2:*** This query finds the employee ID and email address from the table "employee."

```
db.Employee.find({}, { EmployeeID: 9, Email: employee9@example.com })
```




***Query 3:*** This query finds the inventory stock level of different items in the items table.

```
db.InventoryStock.find({}, {ItemID: 5, CurrStockLevel: 81})
```



***Query 4:*** This query is the MongoDB implementation of MySQL query 7. This query calculates the total amount spent by each customer grouped by their payment card type and orders the results from highest to lowest spender.

```
db.ordertable.aggregate([
    {
        $lookup: {
            from: "customer",
            localField: "CustomerID",
            foreignField: "CustomerID",
            as: "customer_info"
        }
    },
    {
        $unwind: "$customer_info"
    },
    {
        $lookup: {
            from: "pay_method",
            localField: "PaymentMethodID",
            foreignField: "PayMethodID",
            as: "payment_info"
        }
    },
    {
        $unwind: "$payment_info"
    },
    {
        $group: {
            _id: {
                CustomerID: "$CustomerID",
                CardType: "$payment_info.CardType"
            },
            TotalSpent: { $sum: "$TotalCost" },
            CustomerName: { $first: "$customer_info.Name" }
        }
    },
    {
        $sort: {
            TotalSpent: -1
        }
    },
    {
        $project: {
            _id: 0,
            CustomerName: "$CustomerName",
            CardType: "$_id.CardType",
            TotalSpent: 1
        }
    }
]);
```

**Stage 1** $lookup

```
{
  from: "customer",
  localField: "CustomerID",
  foreignField: "CustomerID",
  as: "customer_info"
}
```

Output after $lookup stage (Sample of 10 documents)

```
_id: ObjectId('660f5839e1e955df4e325cea')
OrderID : 1
CustomerID : 10
OrderDateTime : 2022-01-19T00:26:06.000+00:00
PayMethodID : "1"
TotalCost : 39.47
▸ Items : Array (1)
▸ customer_info : Array (empty)
```

```
_id: ObjectId('660f5839e1e955df4e325ceb
OrderID : 2
CustomerID : 2
OrderDateTime : 2022-04-07T23:41:28.000+
PayMethodID : "4"
TotalCost : 71.23
▸ Items : Array (1)
▸ customer_info : Array (empty)
```

**Stage 2** $unwind

```
{
    path: "$customer_info",
    preserveNullAndEmptyArrays: true
}
```

Output after $unwind stage (Sample of 10 documents)

```
_id: ObjectId('660f5839e1e955df4e325cea')
OrderID : 1
CustomerID : 10
OrderDateTime : 2022-01-19T00:26:06.000+00:00
PayMethodID : "1"
TotalCost : 39.47
▸ Items : Array (1)
```

```
_id: ObjectId('660f5839e1e955df4e325ceb
OrderID : 2
CustomerID : 2
OrderDateTime : 2022-04-07T23:41:28.000+
PayMethodID : "4"
TotalCost : 71.23
▸ Items : Array (1)
```

**Stage 3** $lookup

```
{
  from: "pay_method",
  localField: "PaymentMethodID",
  foreignField: "PayMethodID",
  as: "payment_info"
}
```

Output after $lookup stage (Sample of 10 documents)

```
_id: ObjectId('660f5839e1e955df4e325cea')
OrderID : 1
CustomerID : 10
OrderDateTime : 2022-01-19T00:26:06.000+00:00
PayMethodID : "1"
TotalCost : 39.47
▸ Items : Array (1)
▸ payment_info : Array (empty)
```

```
_id: ObjectId('660f5839e1e955df4e325ceb
OrderID : 2
CustomerID : 2
OrderDateTime : 2022-04-07T23:41:28.000+
PayMethodID : "4"
TotalCost : 71.23
▸ Items : Array (1)
▸ payment_info : Array (empty)
```

**Stage 4** $unwind

```
{
    path: "$payment_info",
    preserveNullAndEmptyArrays: true  //
}
```

Output after $unwind stage (Sample of 10 documents)

```
_id: ObjectId('660f5839e1e955df4e325cea')
OrderID : 1
CustomerID : 10
OrderDateTime : 2022-01-19T00:26:06.000+00:00
PayMethodID : "1"
TotalCost : 39.47
▸ Items : Array (1)
```

```
_id: ObjectId('660f5839e1e955df4e325ceb
OrderID : 2
CustomerID : 2
OrderDateTime : 2022-04-07T23:41:28.000+
PayMethodID : "4"
TotalCost : 71.23
▸ Items : Array (1)
```

**Stage 5** $group

```
{
  _id: {
    CustomerID: "$CustomerID",
    CardType: "$payment_info.CardType"
  },
  TotalSpent: { $sum: "$TotalCost" },
  CustomerName: { $first: "$customer_info
}
```

Output after $group stage (Sample of 5 documents)

```
▸ _id: Object
TotalSpent : 653.56
CustomerName : null
```

```
▸ _id: Object
TotalSpent : 188.88
CustomerName : null
```

**Stage 6** $sort

```
{
  TotalSpent: -1
}
```

Output after $sort stage (Sample of 5 documents)

```
▸ _id: Object
TotalSpent : 653.56
CustomerName : null
```

```
▸ _id: Object
TotalSpent : 498.70
CustomerName : null
```

**Stage 7** $project

```
{
  _id: 0,
  CustomerName: "$CustomerName",
  CardType: "$_id.CardType",
  TotalSpent: 1
}
```

Output after $project stage (Sample of 5 documents)

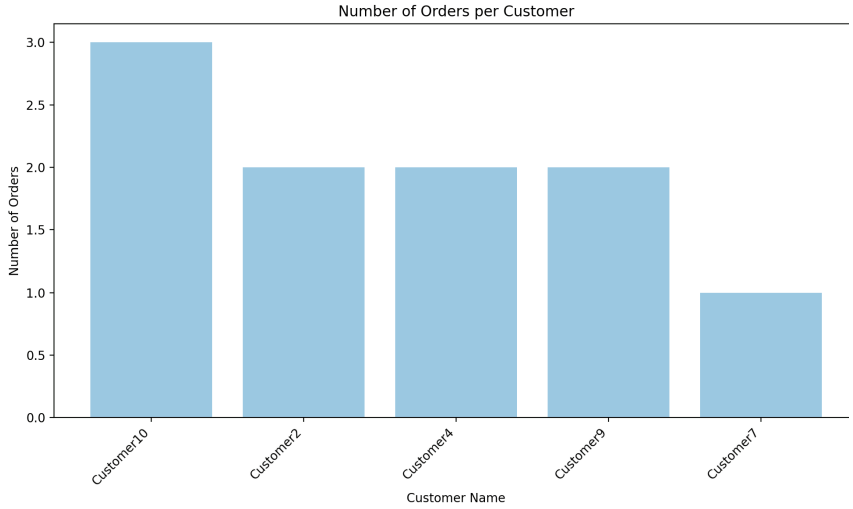
```
TotalSpent : 653.56
CustomerName : null
```

```
TotalSpent : 498.70
CustomerName : null
```

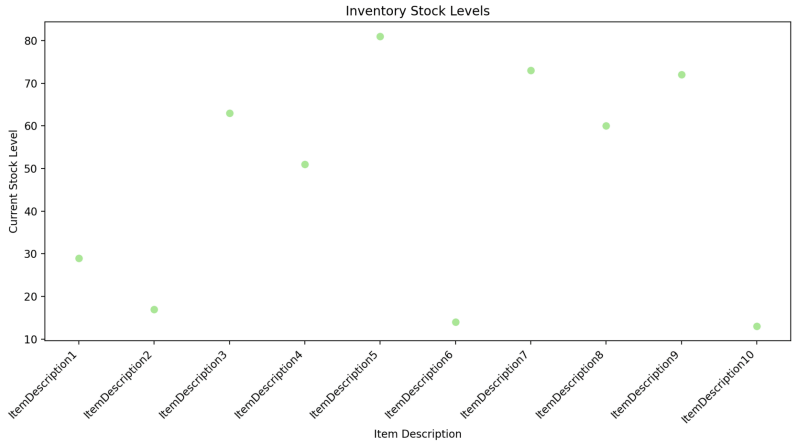
## VI. Database Access via Python3

Connecting Python applications to MySQL is beneficial to our database for various reasons. By connecting Python to our database, we can harness different Python libraries used for data analytics, such as Pandas, Numpy, and Matplotlib, to analyze and visualize data to create helpful insights. Python is also an easy-to-use program that also offers flexibility. The flexibility in the creation of custom code in Python will also allow us to code based on the needs of the business.

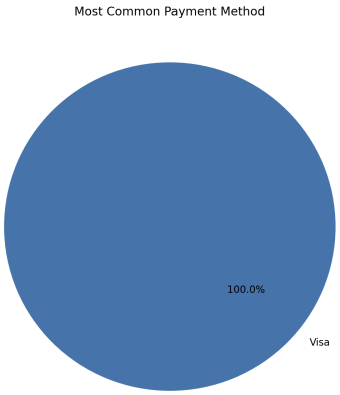
Number of Orders per Customer



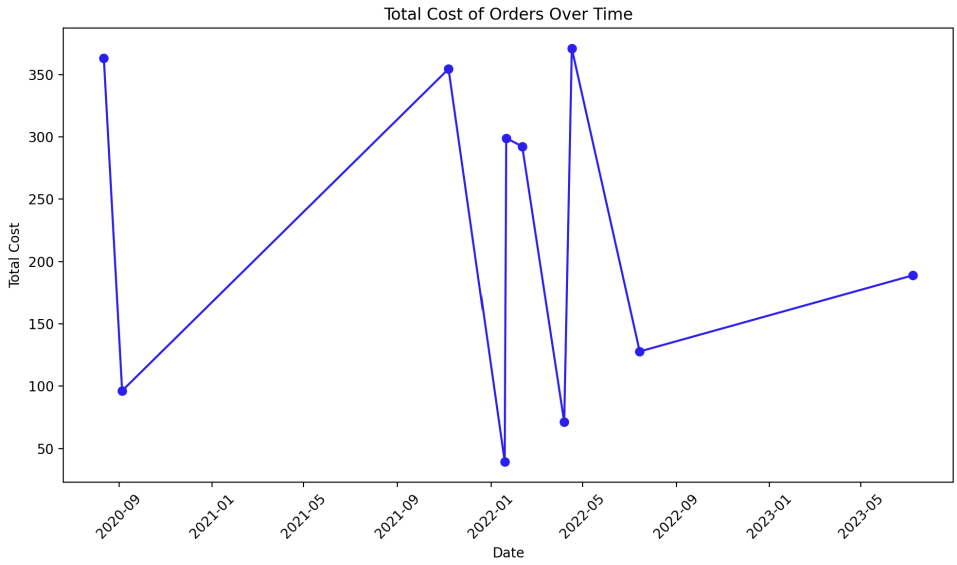

Inventory Stock Levels



Most Common Payment Method




Total Cost of Orders Over Time




## VII. Summary and Recommendation

In this project, we were successfully able to create a ready-to-use database for Jitters. The database implementation has two-fold benefits for Jitters: Jitters will have a robust DBMS to track and store crucial data such as customer information and inventory tracking processes. Furthermore, Jitters will be able to use data in analytics and data visualization, which will greatly assist in making business decisions.

In the future, Jitters can implement an inventory ordering system directly through this database system. This can be done by executing SQL statements within a transaction. Jitters may also develop a front-end interface system to create a user-friendly platform to place orders.

This project challenged us to learn critical information about the ins and outs of database creation and implementation in different DBMS systems. Throughout the project, we learned how to troubleshoot, debug, and think like a business owner, employee, and customer. Ultimately, we were able to implement the lessons learned throughout the course and create a versatile DBMS for Jitters.

In conclusion, this project was able to solve the proposed business problem while also leaving the foundation to build upon the database. Both MySQL and MongoDB implementations are horizontally scalable, so the business has options to choose their desired DBMS. Moreover, the Python implementation has further expanded the DBMS capabilities, as many creative integrations can be made to this database.

## VIII. References

1. Please see following definitions of entities and attributes:
   **KioskSession:** The customer will enter the store and go to the kiosk to place their order. The kiosk will generate a session ID and track the checkout status of the session. The kiosk will also track the timestamp of any placed order.
   **Customer:** This entity represents an individual who is purchasing an item from Jitters. It stores the contact information of the customer and can be used to send promotional offers or understand the customers' order preferences.
   **Order:** This entity generates an orderID and connects the orderID to the customer through the customerID. The order also collects the payment information for the placed order. The order generates the total cost by adding the individual items. The information about the individual items (add-ons, if any) is retrieved from the **OrderItem** entity.
   **PayMethod:** Stores information on how customers are allowed to pay (i.e., cash, various types of credit cards). A customer will be asked to pay via a provided payment method to complete their order.
   **Employee:** Contains information about baristas who work for Jitters. It includes their contact information. An employee with the applicable shift is randomly assigned to complete an order that has been placed by a customer.
   **OrderItem:** Details each item within an order, including quantity, price, and any customizations. This entity allows for a granular look at what exactly a customer has ordered.
   **Item:** Describes the items that are available for sale (i.e., iced coffee, iced tea, hot coffee). This includes a price, description, and possibly any other attributes that can help manage the inventory and present information to the customers (i.e., what would be listed on the actual menu).
   **InventoryStock:** This entity tracks stock levels of items for sale. The inventory stock entity entails all the supplies, consumables, perishables, etc. necessary for the cafe.
   **Supplier:** The supplier entity represents a person(s) or a company(s) who supply certain inventories to the cafe.
   **SupplierOrder:** A supplier order is an order that the cafe places with a specific supplier. This entity stores information such as unit cost, total cost, order date, delivery date, etc.

2. This code accesses JittersDB and performs analytics on it using key libraries like Pandas and Matplotlib.

```
import mysql.connector
from mysql.connector import Error
import pandas as pd
import matplotlib.pyplot as plt

def fetch_data(query):
          """This is out utility function to fetch data from the Jitters database."""
          try:
          connection = mysql.connector.connect(
          host='localhost',
          database='JittersDB',
          user='[username]',
          password='[password]'
          )
          if connection.is_connected():
          df = pd.read_sql(query, connection)
          return df
          except Error as e:
          print(f"Error: {e}")
          finally:
          if connection and connection.is_connected():
          connection.close()

def plot_number_of_orders_per_customer():
          """Plots the number of orders per customer."""
          query = """
          SELECT c.Name, COUNT(o.OrderID) AS OrderCount
          FROM Customer c
          JOIN OrderTable o ON c.CustomerID = o.CustomerID
          GROUP BY c.CustomerID
          ORDER BY OrderCount DESC;
          """
          df = fetch_data(query)
          plt.figure(figsize=(10, 6))
          plt.bar(df['Name'], df['OrderCount'], color='skyblue')
          plt.xlabel('Customer Name')
          plt.ylabel('Number of Orders')
          plt.xticks(rotation=45, ha="right")
          plt.title('Number of Orders per Customer')
          plt.tight_layout()
          plt.show()

def plot_inventory_stock_levels_scatter():
          """Plots inventory stock levels as a scatter plot."""
          query = """
          SELECT Description, CurrStockLevel
          FROM Item
          JOIN InventoryStock ON Item.ItemID = InventoryStock.ItemID;
          """
          df = fetch_data(query)
          plt.figure(figsize=(10, 6))
          plt.scatter(df['Description'], df['CurrStockLevel'], color='lightgreen')
          plt.xlabel('Item Description')
          plt.ylabel('Current Stock Level')
          plt.xticks(rotation=45, ha="right")
          plt.title('Inventory Stock Levels')
          plt.tight_layout()
          plt.show()

def plot_most_common_payment_method():
          """Plots the most common payment method."""
          query = """
          SELECT CardType, COUNT(*) AS Count
```

```
            FROM PAY_METHOD
            GROUP BY CardType
            ORDER BY Count DESC;
            """
            df = fetch_data(query)
            plt.figure(figsize=(8, 8))
            plt.pie(df['Count'], labels=df['CardType'], autopct='%1.1f%%', startangle=140)
            plt.title('Most Common Payment Method')
            plt.show()

def plot_total_cost_of_orders_over_time():
            """Plots total cost of orders over time as a line graph."""
            query = """
            SELECT DATE(OrderDateTime) AS OrderDate, SUM(TotalCost) AS DailyTotalCost
            FROM OrderTable
            GROUP BY OrderDate
            ORDER BY OrderDate;
            """
            df = fetch_data(query)
            plt.figure(figsize=(10, 6))
            plt.plot(df['OrderDate'], df['DailyTotalCost'], marker='o', linestyle='-', color='blue')
            plt.xlabel('Date')
            plt.ylabel('Total Cost')
            plt.title('Total Cost of Orders Over Time')
            plt.xticks(rotation=45)
            plt.tight_layout()
            plt.show()

if __name__ == "__main__":
            plot_number_of_orders_per_customer() # Bar graph
            plot_inventory_stock_levels_scatter()  # Scatter plot
            plot_most_common_payment_method() # Pie chart
            plot_total_cost_of_orders_over_time()  # Line graph
```

3. We created a data transfer Python script to load all of our generated data from MySQL directly into MongoDB Compass. For this script, we utilized Pymongo to use Python in conjunction with MongoDB. Please see below for the script we created and ran:

```
import mysql.connector
from pymongo import MongoClient
from faker import Faker
import random
from bson.decimal128 import Decimal128
import decimal

fake = Faker() # Initializes our Faker for additional random data generation (only if needed)

# MySQL connection setup
mysql_conn = mysql.connector.connect(
    host='localhost',
    user='username',
    password='password',
    database='JittersDB'
)

# MongoDB connection setup
mongo_client = MongoClient('mongodb://localhost:27017/')
mongo_db = mongo_client["JittersDB"]

def convert_decimal_fields(document):
    for key, value in document.items():
        if isinstance(value, decimal.Decimal):
```

```
                document[key] = Decimal128(str(value))
        return document

    def transfer_data(table_name, mongo_collection):
        mysql_cursor = mysql_conn.cursor(dictionary=True)
        query = f"SELECT * FROM {table_name}"
        mysql_cursor.execute(query)
        data = mysql_cursor.fetchall()
        for document in data:
            document = convert_decimal_fields(document)
            # Additional logic for the nested items in orders
            if table_name == "OrderTable":
                order_items_cursor = mysql_conn.cursor(dictionary=True)
                order_items_cursor.execute("SELECT * FROM ORDERITEM WHERE OrderID = %s", (document["OrderID"],))
                order_items = order_items_cursor.fetchall()
                document["Items"] = [convert_decimal_fields(item) for item in order_items]
            mongo_collection.insert_one(document)

    if __name__ == "__main__":
        transfer_data("Customer", mongo_db.customers)
        transfer_data("Item", mongo_db.items)
        transfer_data("Supplier", mongo_db.suppliers)
        transfer_data("PAY_METHOD", mongo_db.pay_methods)
        transfer_data("OrderTable", mongo_db.orders)
        transfer_data("Customer_Contact_Info", mongo_db.customer_contact_info)
        transfer_data("InventoryStock", mongo_db.inventory_stock)
        transfer_data("SupplierOrder", mongo_db.supplier_orders)
        transfer_data("EmployeeWorksOrder", mongo_db.employee_works_order)
        transfer_data("OrderItemhasItem", mongo_db.order_item_has_item)
        transfer_data("ItemSupplier", mongo_db.item_supplier)
        transfer_data("Kiosk_Session", mongo_db.kiosk_session)
        print("Data transfer complete.")

    # Closes MySQL connection
    mysql_conn.close()
```