SQL and NoSQL Database Assessment

School of Engineering, Arts, Science and Technology

Kicks Ltd. Database

Student Number: S267562

Word count: 3,023

Table of Contents

LIST OF FIGURES

List of Tables

Introduction and Background

This report addresses the fundamental database needs of Kicks Ltd, a start-up footwear retailer. It chronicles the creation of a sales database, tracing its inception from the initial 49 transactions and showcasing the fluid interaction between customers and employees during order processing. Going beyond, it outlines the meticulous process of conceptualization, logical structuring, and physical implementation of the databases. Additionally, it explores vital aspects including data protection, management, update flexibility, and scalability.

Mission Statement

The database serves to empower the business by providing robust support for maintaining, analyzing, and optimizing sales dynamics encompassing employees, customers, products, and orders. It enables seamless and secure product ordering by customers and processing by employees, while equipping stakeholder with actionable insights for informed decision-making regarding top-selling brands and footwear categories.

Mission Objective

1. Implementing CRUD functionalities for store data management.
2. Generating insightful queries on orders and customer data.
3. Tracking demand for products, brands, and footwear categories.
4. Monitoring staff performance for identifying top-performing employees.
5. Ensuring privacy and security of sensitive employee details.
6. Ensuring store data accuracy and relevance.
7. Providing efficient data access for stakeholders, employees, and customers through comprehensive querying.

Relational Database Design

Users and Tasks

Table 1 summarizes the anticipated user interactions with the database.

Table 1 Users and their tasks

Users	**Tasks**
Store (stakeholders)	- Execute queries and derive insights utilizing the SQL database. - Perform input, update, modification, and deletion of data within the SQL database as needed. - Incorporate new data into the database as needed. - Grant and modify privileges as needed.
Employees	- Monitor and manage order status. - Record and analyze product sales. - Void transactions where required.

Customer	- Placing Orders: to browse available products, select items they wish to purchase, and place orders. - Reviewing Past Purchases: Customers can review their purchase history, including details of past orders and products bought, to facilitate reordering or reference purposes.

California Consumer Privacy Act (CCPA)

CCPA is a state statute aimed at protecting privacy and consumer rights for residents in California, USA. Given that Kicks will be handling personal data, compliance with this legislation is essential to safeguard individuals' rights. This is especially pertinent if the company intends to retain employees' national insurance (NI) information for tax classification purposes. It's crucial to prioritize data protection and security considerations from the outset, as retrofitting these measures later can be challenging.

Business Rule

1. Each footwear item purchased by a customer is categorized into a distinct order. Consequently, each purchased shoe is treated as an individual order, each assigned a unique orderID.
2. Only one employee is assigned to process an order, although collaboration among employees is permitted. Ultimately, the employee who finalizes the order will have their employee ID associated with it.
3. The store necessitates an in-store sales-specific database tailored exclusively to transactions occurring within the physical store. This database focuses solely on interactions among employees, customers, and purchased products (recorded as orders), without encompassing any involvement or details related to third-party entities such as supply companies, inventory management, or delivery services.
4. All attributes are mandated to have the "NOT NULL" constraints.

Kicks Requirements: Reviewing the current system.

Currently, Kicks manages its sales data using an Excel spreadsheet, which lacks a streamlined approach for generating insightful and actionable queries.

Table 2 The Current System

	A	B	C	D	E
1	**Store**	**Employee**	**Customer**	**Product**	**Order**
2	storeID	employeeID	customerID	productCode	orderID
3	storeName	employeeName	customerName	brandID	productCode
4	storeAddress	employeePosition	customerAddress	brandName	customerID
5	storeEmail	NI	customerEmail	categoryType	employeeID
6	storeTelephone	contract	customerTelephone	categoryName	orderDate
7	storeOpenDate	taxCode	deliveryAddress	colour	orderStatus
8	numberOfEmployees	salary		gender	
9		employeeAddress		size	
10		employeeEmail		price	
11		employeeTelephone			

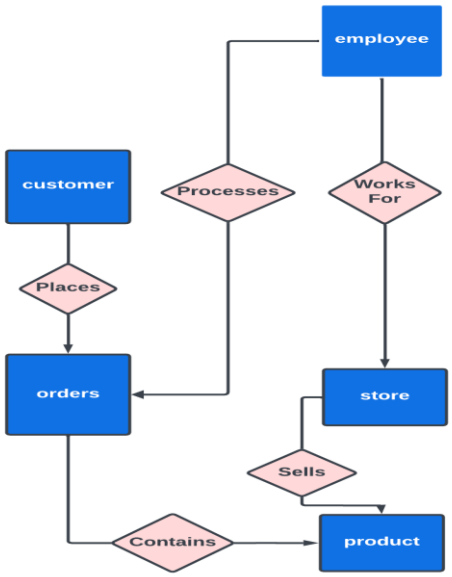
To handle its growing sales data efficiently, Kicks plans to transition from Excel spreadsheets to a Relational Database Management System (RDBMS), commonly known as an SQL database. RDBMS organizes data into structured tables, enabling seamless management and access. SQL, the standardized query language used in RDBMS, facilitates complex queries through a unified interface (Sahatqija et al., 2018). With SQL, users can insert, query, update, secure, comply with CCPA and delete data effortlessly, making it ideal for data manipulation and retrieval tasks.

Database Design Decisions

Conceptual Design

A conceptual design was derived from an analysis of the daily sales activities observed at the store, the excel sheet and explanations from the stakeholders. This design reflects the process by which customers purchase footwear products through placing orders at the store. Additionally, it encompasses the responsibility of the store's staff to process these orders as part of their duties.

Figure 1 Conceptual Design



The design visually depicts entities such as stores, products, employees, customers, and orders, represented by blue blocks. These entities are interconnected by lined arrows, illustrating their relationships. Additionally, pink diamonds show the nature of the relationships between the entities.

Normalisation of the data stored in the Excel sheets was essential to proceed with the conceptual design. Normalisation aims to reduce redundancy and prevent anomalies related to data insertion, deletion, and updating (Cvetkov-Iliev, 2023).

1st Normal Form

To attain First Normal Form (1NF), it is essential that attribute domains consist solely of atomic, singular values (Terai, 2023). Currently, the entities for store, employee, and customer do not adhere to the First Normal Form (1NF) because attributes such as address, name, and contract allow for multiple values.

Table 3 First Normal Form

	A	B	C	D	E
1	**Store**	**Employee**	**Customer**	**Product**	**Orders**
2	storeID	employeeID	customerID	productCode	orderID
3	storeName	employeeFirstName	customerFirstName	brandID	productCode
4	storeBuildingNumber	employeeLastName	customerLastName	brandName	customerID
5	storeStreet	employeePosition	customerBuildingNumber	categoryType	employeeID
6	storeCity	NI	customerStreet	categoryName	orderDate
7	storeState	employeeStartDate	customerCity	colour	orderStatus
8	storePostCode	employeeEndDate	customerState	gender	
9	storeEmail	taxCode	customerPostCode	size	
10	storeTelephone	salary	customerEmail	price	
11	storeOpenDate	employeeBuildingNumber	customerTelephone		
12	numberOfEmployees	employeeStreet			
13		employeeCity			
14		employeeState			
15		employeePostCode			
16		employeeEmail			
17		employeeTelephone			

As depicted in blue in table 3, the attributes address, name, and contract are disaggregated into individual atomic values within the store, employee, and customer entities, respectively.

2nd Normal Form

To achieve Second Normal Form (2NF), i ensure that tables adhere to First Normal Form (1NF) and that non-key attributes are dependent on the entire primary key (Ferreira L.M. et al., 2023).

In our dataset:

- In the employee table, attributes like CONTRACT DURATION, TAX CODE, and SALARY don't solely depend on the entire primary key. Thus, i create the employeePayroll table to resolve this issue.
- Similarly, in the Product table, brandName depends on brandID, violating 2NF. I introduce a brand table and kept brandID as a foreign key in the Product table.

Table 4 Second Normal Form

	A	B	C	D	E	F	G
1	**Store**	**Employee**	**Employee Payroll**	**Customer**	**Product**	**Brand**	**Orders**
2	storeID	employeeID	NI	customerID	productCode	brandID	orderID
3	storeName	employeeFirstName	employeeID	customerFirstName	brandID	brandName	productCode
4	storeBuildingNumber	employeeLastName	employeeStartDate	customerLastName	categoryType		customerID
5	storeStreet	employeePosition	employeeEndDate	customerBuildingNumber	categoryName		employeeID
6	storeCity	employeeBuildingNumber	taxCode	customerStreet	colour		orderDate
7	storeState	employeeStreet	salary	customerCity	gender		orderStatus
8	storePostCode	employeeCity		customerState	size		
9	storeEmail	employeeState		customerPostCode	price		
10	storeTelephone	employeePostCode		customerEmail			
11	storeOpenDate	employeeEmail		customerTelephone			
12	numberOfEmployees	employeeTelephone					

The attributes NI, employeeStartDate, employeeEndDate, taxCode, and salary have been moved from the employee table to a new table called employeePayroll, while brandID and brandName have been transferred from the product table to a new table named brand.

3rd Normal Form

Third Normal Form (3NF) eliminates transitive dependencies, ensuring data integrity and reducing redundancy (Rajendran R.K, 2023). To achieve 3NF:

1. Ensure the table is already in Second Normal Form.
2. Remove any transitive dependencies, where non-prime attributes depend on other non-prime attributes.

In our case, the product table contains transitive dependencies as categoryName depends on categoryID, which itself depends on productCode.

Table 5 Third Normal Form

	A	B	C	D	E	F	G	H
1	**Store**	**Employee**	**Employee Payroll**	**Customer**	**Product**	**Brand**	**Category**	**Orders**
2	storeID	employeeID	NI	customerID	productCode	brandID	categoryType	orderID
3	storeName	employeeFirstName	employeeID	customerFirstName	brandID	brandName	categoryName	productCode
4	storeBuildingNumber	employeeLastName	employeeStartDate	customerLastName	categoryType			customerID
5	storeStreet	employeePosition	employeeEndDate	customerBuildingNumber	colour			employeeID
6	storeCity	employeeBuildingNumber	taxCode	customerStreet	gender			orderDate
7	storeState	employeeStreet	salary	customerCity	size			orderStatus
8	storePostCode	employeeCity		customerState	price			
9	storeEmail	employeeState		customerPostCode				
10	storeTelephone	employeePostCode		customerEmail				
11	storeOpenDate	employeeEmail		customerTelephone				
12	numberOfEmployees	employeeTelephone						

The categoryType and categoryName attributes have been shifted from the product table to a new category table, fulfilling Third Normal Form (3NF) criteria.

Relationships

Table 6 Table Relationships

Table	Relationship with →	Table
store	ONE to MANY The ONE store sells MANY products	Product
product	MANY to ONE MANY products can be produced by ONE brand	Brand
product	MANY to ONE MANY products belong to ONE category	Category
store	ONE to Many The ONE store employs MANY Staffs	Employee
employee	MANY to ONE MANY employees have ONE payroll	employeePayroll
customer	ONE to MANY ONE customer can place MANY orders	orders
employee	ONE to MANY ONE employee can process MANY orders	orders
orders	ONE TO MANY ONE order can contain MANY products	product

Cardinality

Cardinality refers to the numerical relationship between entities in a database, categorized as one-to-one, one-to-many, many-to-one, or many-to-many (Link .S. et al.,2023) with my decided cardinality explained below.

Store to Product: Each store sells multiple products, with each product exclusively sold by one store.

Product to Brand: Each product is associated with one brand, while a brand can produce multiple types of products.

Product to Category: Each product belongs to one category, while a category can encompass multiple products.

Store to Employee: Each store employs multiple employees, with each employee working exclusively for one store.

Employee to EmployeePayroll: Each employee has payroll information, with payroll information related to one employee.

Customer to Orders: Each customer places multiple orders, with each order placed by only one customer.

Employee to Orders: An employee processes multiple orders, with each order processed by only one employee.

Orders to Products: Each order contains multiple products, with each product included in multiple orders.

Table Attributes & Constraints

Table 7 shows the attributes and constraints in the dataset.

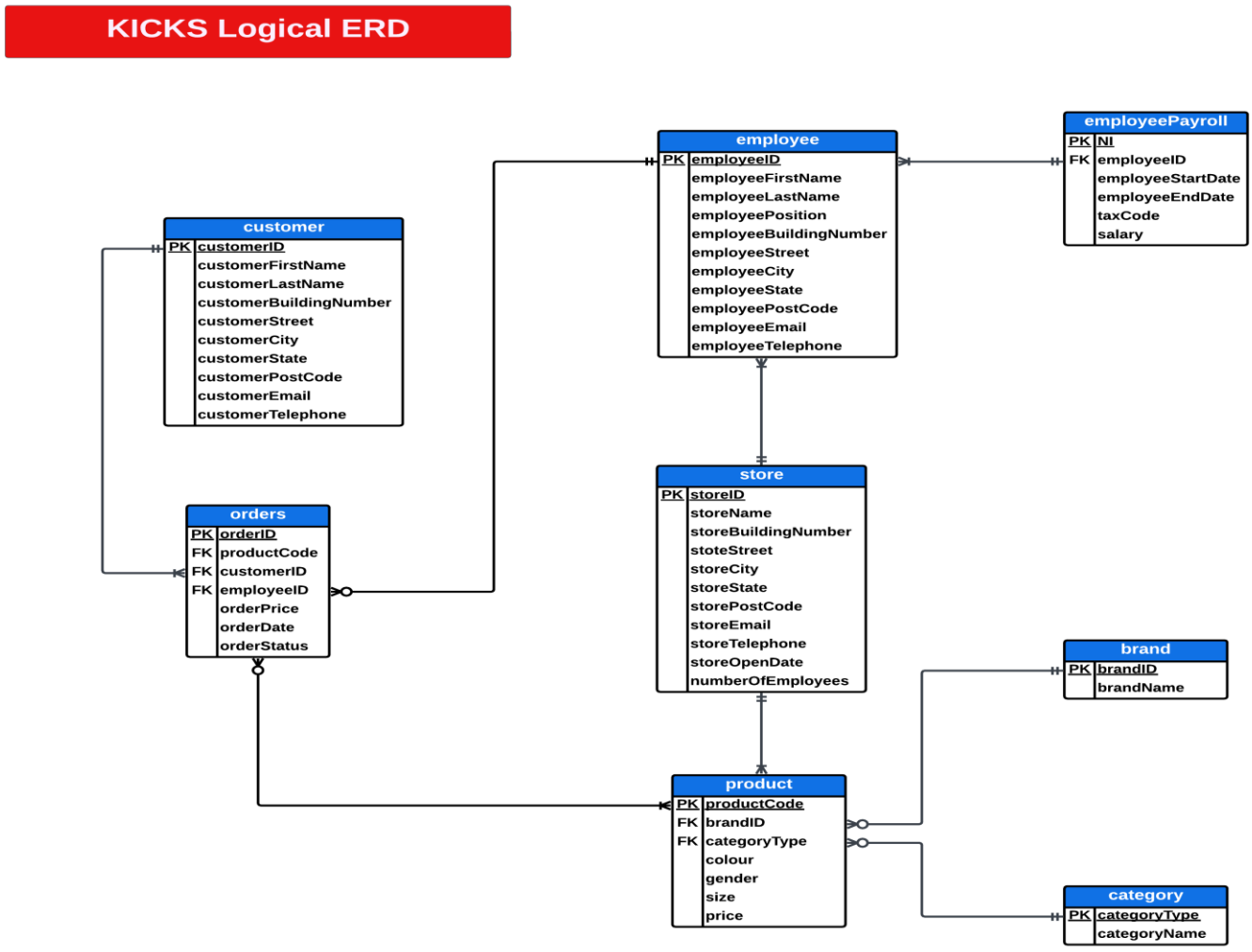
Table 7 Attributes and Constraints

tableTitle	attributesTitles	key	dataType (length)	constraints
store	storeID	Primary Key	VARCHAR (55)	UNIQUE
	storeName	Non-Prime	VARCHAR (55)	NOT NULL
	storeBuildingNumber	Non-Prime	VARCHAR (55)	NOT NULL
	storeStreet	Non-Prime	VARCHAR (55)	NOT NULL
	storeCity	Non-Prime	VARCHAR (55)	NOT NULL
	storeState	Non-Prime	VARCHAR (55)	NOT NULL
	storePostCode	Non-Prime	VARCHAR (55)	NOT NULL
	storeEmail	Non-Prime	VARCHAR (55)	NOT NULL
	storeTelephone	Non-Prime	VARCHAR (55)	NOT NULL
	storeOpenDate	Non-Prime	VARCHAR (55)	NOT NULL
	numberOfEmployees	Non-Prime	INTEGER	NOT NULL
employee	employeeID	Primary Key	VARCHAR (55)	UNIQUE
	employeeFirstName	Non-Prime	VARCHAR (55)	NOT NULL
	employeeLastName	Non-Prime	VARCHAR (55)	NOT NULL
	employeePosition	Non-Prime	VARCHAR (55)	NOT NULL
	employeeBuildingNumber	Non-Prime	VARCHAR (55)	NOT NULL
	employeeStreet	Non-Prime	VARCHAR (55)	NOT NULL
	employeeCity	Non-Prime	VARCHAR (55)	NOT NULL
	employeeState	Non-Prime	VARCHAR (55)	NOT NULL
	employeePostCode	Non-Prime	VARCHAR (55)	NOT NULL
	employeeEmail	Non-Prime	VARCHAR (55)	NOT NULL
	employeeTelephone	Non-Prime	VARCHAR (55)	NOT NULL
employeePayroll	NI	Primary Key	VARCHAR (55)	UNIQUE
	employeeID	Foreign Key	VARCHAR (55)	NOT NULL
	employeeStartDate	Non-Prime	VARCHAR (55)	NOT NULL
	employeeEndDate	Non-Prime	VARCHAR (55)	NOT NULL
	taxCode	Non-Prime	VARCHAR (55)	NOT NULL
	Salary	Non-Prime	VARCHAR (55)	NOT NULL
customer	customerID	Primary Key	VARCHAR (55)	UNIQUE
	customerFirstName	Non-Prime	VARCHAR (55)	NOT NULL
	customerLastName	Non-Prime	VARCHAR (55)	NOT NULL
	customerBuildingNumber	Non-Prime	VARCHAR (55)	NOT NULL
	cusomerStreet	Non-Prime	VARCHAR (55)	NOT NULL
	customerCity	Non-Prime	VARCHAR (55)	NOT NULL
	customerState	Non-Prime	VARCHAR (55)	NOT NULL
	customerPostCode	Non-Prime	VARCHAR (55)	NOT NULL
	customerEmail	Non-Prime	VARCHAR (55)	NOT NULL
	customerTelephone	Non-Prime	VARCHAR (55)	NOT NULL
brand	brandID	Primary Key	VARCHAR (55)	UNIQUE
	brandName	Non-Prime	VARCHAR (55)	NOT NULL
category	categoryType	Primary Key	VARCHAR (55)	UNIQUE
	category Name	Non-Prime	VARCHAR (55)	NOT NULL
product	productCode	Primary Key	VARCHAR (55)	UNIQUE
	brandID	Foreign Key	VARCHAR (55)	NOT NULL
	categoryType	Foreign Key	VARCHAR (55)	NOT NULL
	Colour	Non-Prime	VARCHAR (55)	NOT NULL
	Gender	Non-Prime	VARCHAR (55)	NOT NULL
	Size	Non-Prime	INTEGER	NOT NULL
	Price	Non-Prime	DECIMAL (55, 2)	NOT NULL
orders	orderID	Primary Key	VARCHAR (55)	UNIQUE
	productCode	Foreign Key	VARCHAR (55)	NOT NULL
	customerID	Foreign Key	VARCHAR (55)	NOT NULL
	employeeID	Foreign Key	VARCHAR (55)	NOT NULL
	orderPrice	Non-Prime	DECIMAL (55, 2)	NOT NULL
	orderDate	Non-Prime	VARCHAR (55)	NOT NULL
	orderStatus	Non-Prime	VARCHAR (55)	NOT NULL

Logical Design

The logical design created and represented through ERDs, illustrates the database schema structure, including tables, columns, relationships, and constraints, aligning with system requirements and data relationships, with cardinality details provided (Amran, Mohamed, & Bahry, 2018).

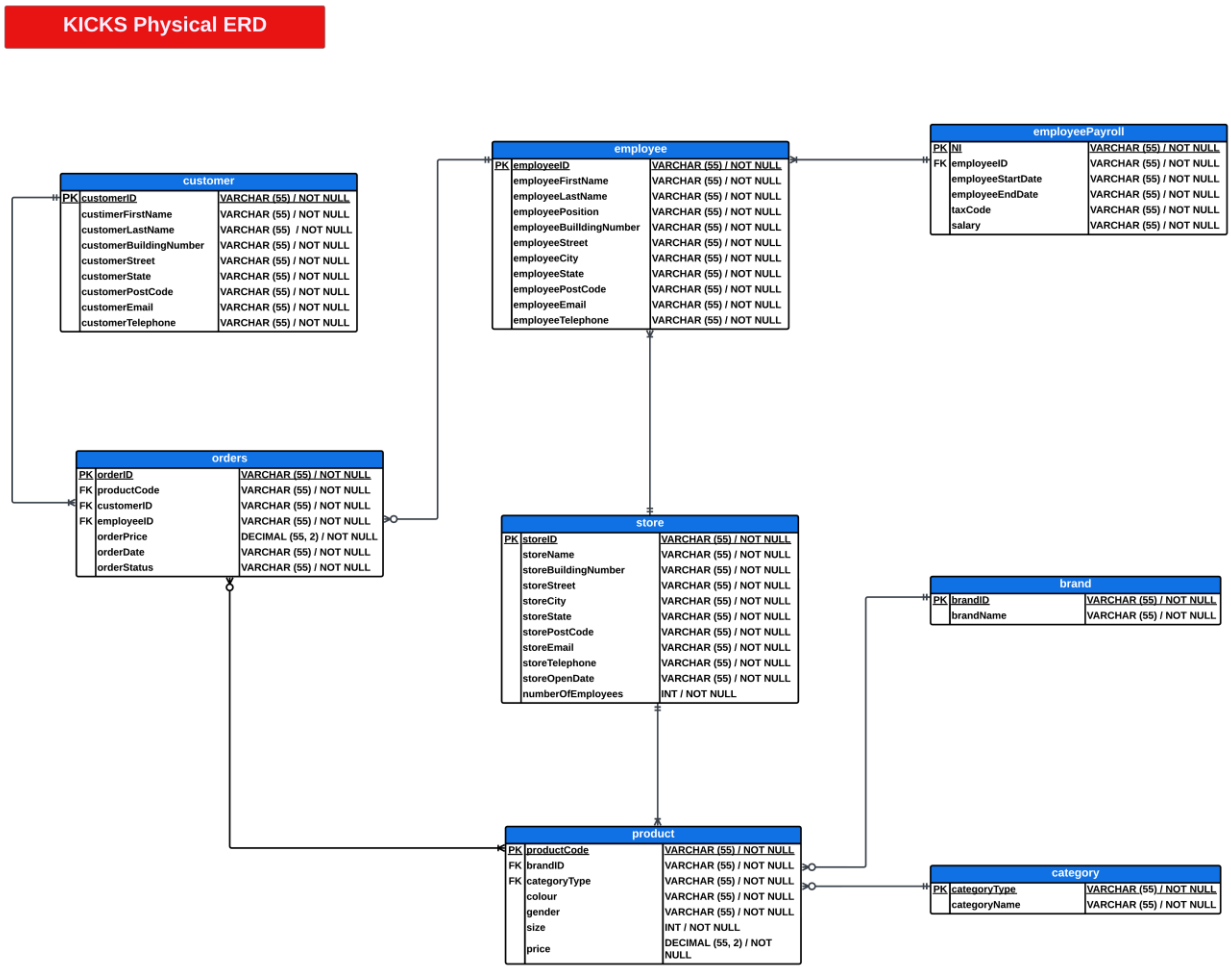
Figure 2 Kicks Logical ERD



Physical Design

The physical design stage transitions from determining "what" will be stored to defining "how" it will be stored. (Connolly and Begg, 2014).

Figure 3 Kicks Physical ERD



The diagram above includes essential elements such as keys, attributes, attribute types, lengths, and constraints. The decisions pertaining to these elements are elaborated below.

Attributes & Lengths

To accommodate diverse global locations and address formats, VARCHAR was chosen as the data type for attributes like "postcode" and "buildingNumber," with a length of 55. Four attributes have distinct variable types, such as "numberOfEmployees" and "Size," which are designated as INTEGER to adhere to business rules. "Price" and "orderPrice" are designated as Decimal to handle decimal figures accurately.

Constraints & Foreign Key

All attributes are mandated to have the "NOT NULL" constraints in alignment with the business rule, ensuring that all data fields must contain information. Moreover, "NOT NULL" constraints serve as a pervasive mechanism for maintaining data integrity. They provide the query optimizer with crucial contextual information, enabling optimizations that markedly reduce query latency (Gartner .M 2023).

To improve performance in foreign-key joins, adding NOT NULL constraints to referenced columns enhances efficiency by limiting the number of scanned and joined rows, thereby reducing query latency.

Potential User Enquiries

Table 8 User Enquiries

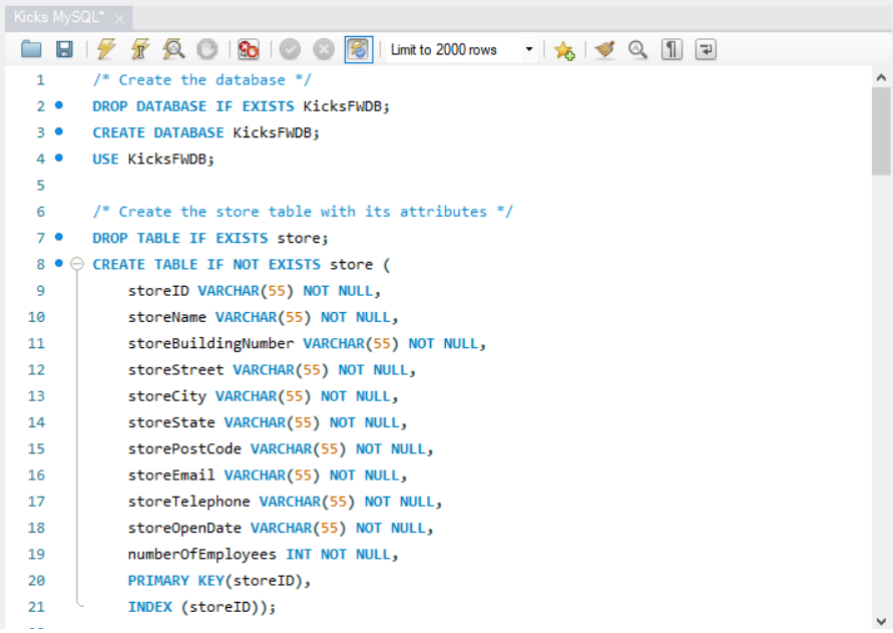
User	Enquiries
Store (Stakeholders)	- What is the stores sales revenue? - What are the emerging trends in customer preference ? - How many employees do i currently have in our retail store ? - What is the average employee work rate (service rate), and are there any patterns? - What measures are in place to ensure data security and confidentiality ? - What are our top 3 performing footwear brands & category names
Employees	- Is the amount of order processing taken into consideration when determining employee rewards? - What are the busiest days ?
Customer	- Can you help me find a sports shoe in my size ? - Can you recommend comfortable casual shoes ? - Can you show me the latest collection of running shoes? - What are the top rated football shoes ? - I am looking for a casual shoe under $50, any recommendations? - Can you provide me with the details of the best selling female sport shoes? - Can I get a black casual shoe for work? - Which brands offer the best football shoes?

Relational Database Implementation

Database Build

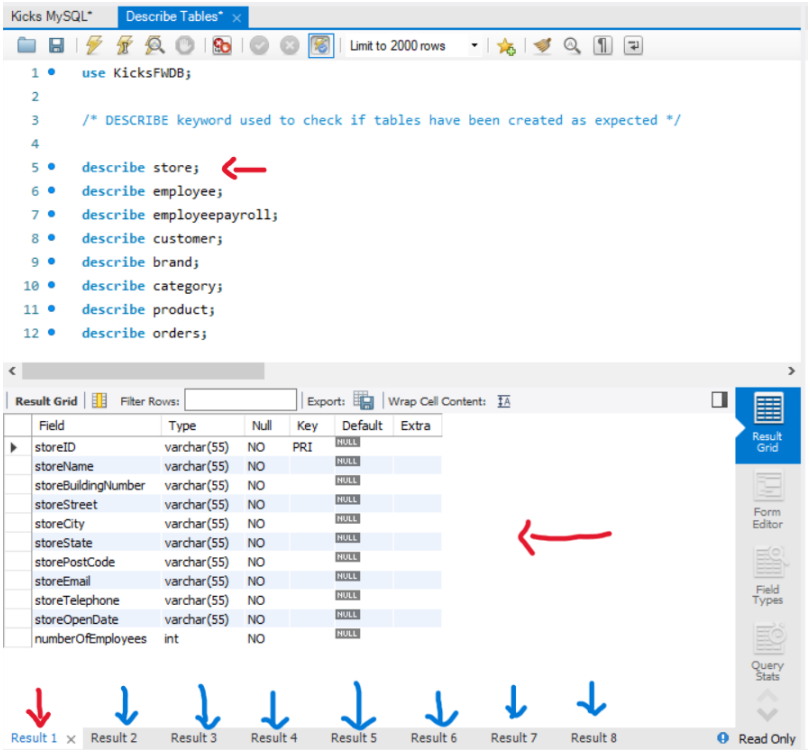
MySQL Workbench was utilized to conduct forward engineering of the physical Entity-Relationship Diagram (ERD) model, thereby generating the script responsible for creating both the database and its associated tables. Indexes were incorporated into the script prior to execution for optimization purposes. Further details regarding their significance will be elaborated in the optimization section of this report.

Figure 4 Table Creation on workbench.



The MySQL DESCRIBE command was employed to furnish a comprehensive overview of a table's structure, encompassing its column names, utilized data types, and any constraints applied to the columns.

Figure 5 Describing the Tables



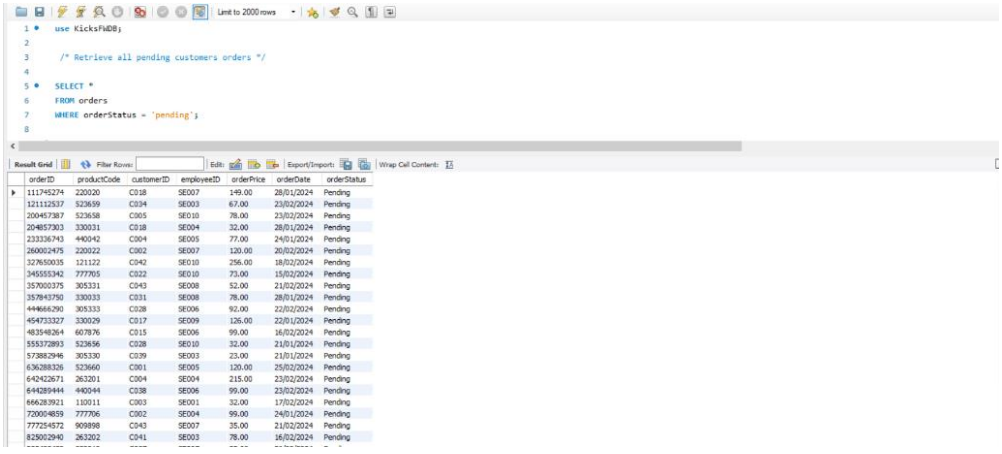
Testing Queries and Generating Insights

Queries and Insights

Multiple queries were executed to evaluate the database and derive useful insights.

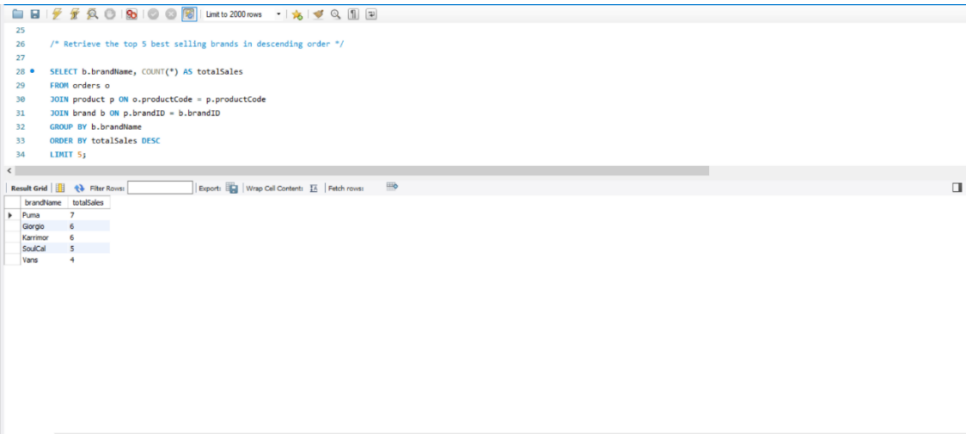
Query 1 Retrieve all pending orders.

Figure 6 Insight on Pending Orders



Query 2 Retrieve the top 5 best-selling brands in descending order.

Figure 7 Insight on Brands.

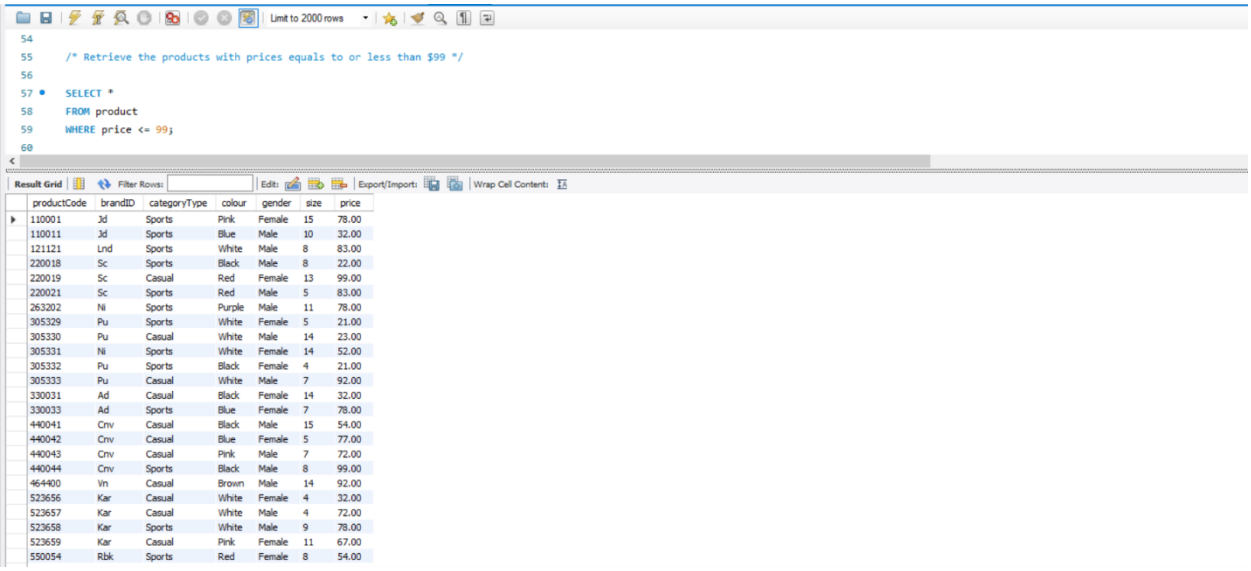


```
/* Retrieve the top 5 best selling brands in descending order */

SELECT b.brandName, COUNT(*) AS totalSales
FROM orders o
JOIN product p ON o.productCode = p.productCode
JOIN brand b ON p.brandID = b.brandID
GROUP BY b.brandName
ORDER BY totalSales DESC
LIMIT 5;
```

brandName	totalSales
Puma	7
Giorgio	6
Karrimor	6
SoulCal	5
Vans	4

Query 3 Retrieves the products with prices equals to or less than $99.

Figure 8 Insight on Products.



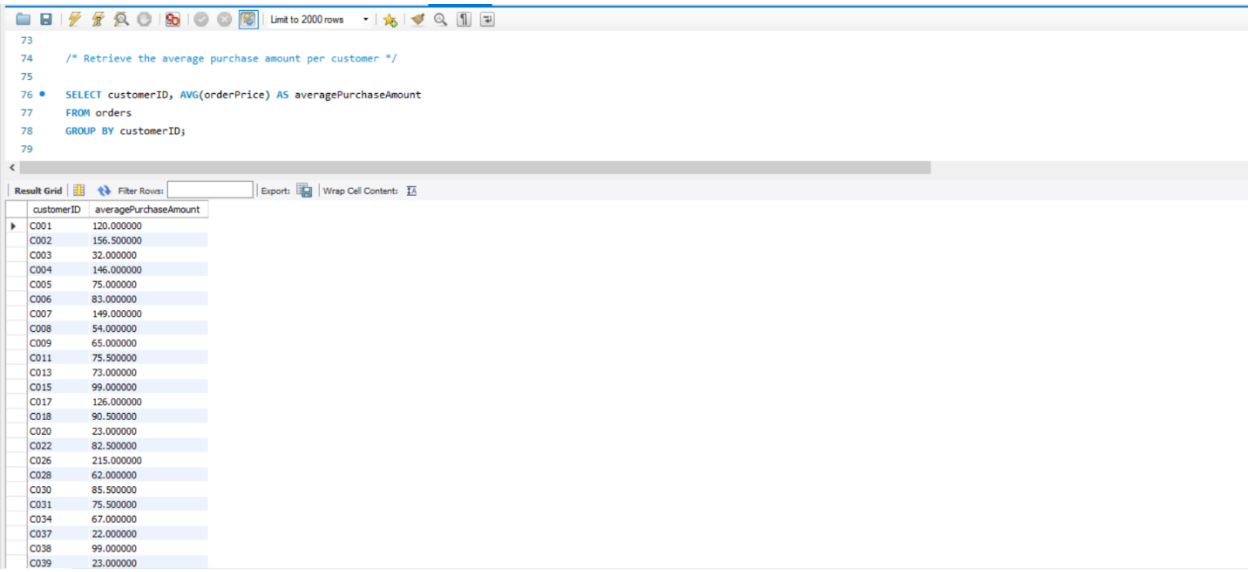
```
/* Retrieve the products with prices equals to or less than $99 */

SELECT *
FROM product
WHERE price <= 99;
```

productCode	brandID	categoryType	colour	gender	size	price
110001	Jd	Sports	Pink	Female	15	78.00
110011	Jd	Sports	Blue	Male	10	32.00
121121	Lnd	Sports	White	Male	8	83.00
220018	Sc	Sports	Black	Male	8	22.00
220019	Sc	Casual	Red	Female	13	99.00
220021	Sc	Sports	Red	Male	5	83.00
263202	Ni	Sports	Purple	Male	11	78.00
305329	Pu	Sports	White	Female	5	21.00
305330	Pu	Casual	White	Male	14	23.00
305331	Ni	Sports	White	Female	14	52.00
305332	Pu	Sports	Black	Female	4	21.00
305333	Pu	Casual	White	Male	7	92.00
330031	Ad	Casual	Black	Female	14	32.00
330033	Ad	Sports	Blue	Female	7	78.00
440041	Cnv	Casual	Black	Male	15	54.00
440042	Cnv	Casual	Blue	Female	5	77.00
440043	Cnv	Casual	Pink	Male	7	72.00
440044	Cnv	Sports	Black	Male	8	99.00
464400	Vn	Casual	Brown	Male	14	92.00
523656	Kar	Casual	White	Female	4	32.00
523657	Kar	Casual	White	Male	4	72.00
523658	Kar	Sports	White	Male	9	78.00
523659	Kar	Casual	Pink	Female	11	67.00
550054	Rbk	Sports	Red	Female	8	54.00

This query will retrieve all rows from the "products" table where the value in the "price" column is less than or equals to 99 with the built in less than or equals to function <=.

Query 4 Figure 9 Retrieves the average purchase amount per customer.

Figure 9 Insights on Customers



```
73
74    /* Retrieve the average purchase amount per customer */
75
76 •  SELECT customerID, AVG(orderPrice) AS averagePurchaseAmount
77    FROM orders
78    GROUP BY customerID;
79
```

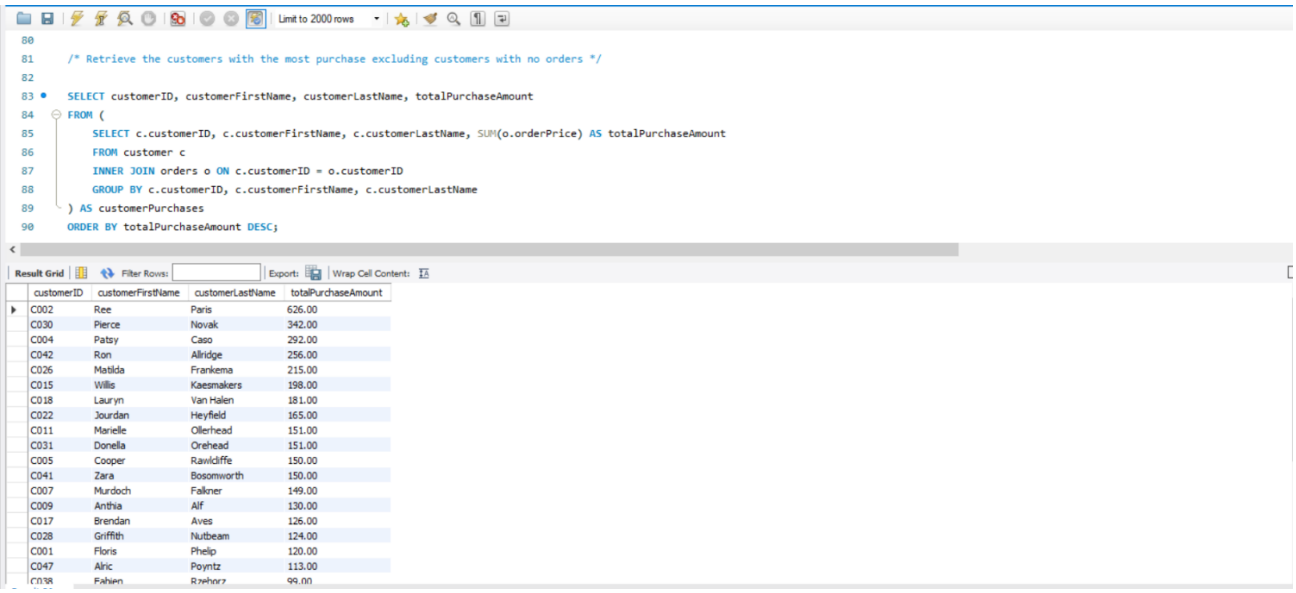
Result Grid | Filter Rows: | Export: | Wrap Cell Contents:

customerID	averagePurchaseAmount
C001	120.000000
C002	156.500000
C003	32.000000
C004	146.000000
C005	75.000000
C006	83.000000
C007	149.000000
C008	54.000000
C009	65.000000
C011	75.500000
C013	73.000000
C015	99.000000
C017	126.000000
C018	90.500000
C020	23.000000
C022	82.500000
C026	215.000000
C028	62.000000
C030	85.500000
C031	75.500000
C034	67.000000
C037	22.000000
C038	99.000000
C039	23.000000

This query selects the "customerID" column from the "orders" table and calculates the average (using the built in average function) purchase amount per customer by grouping the results based on the "customerID" column.

Query 5 Retrieves the customers with the most purchase excluding customers with no orders.

Figure 10 Insights on Customer Purchase



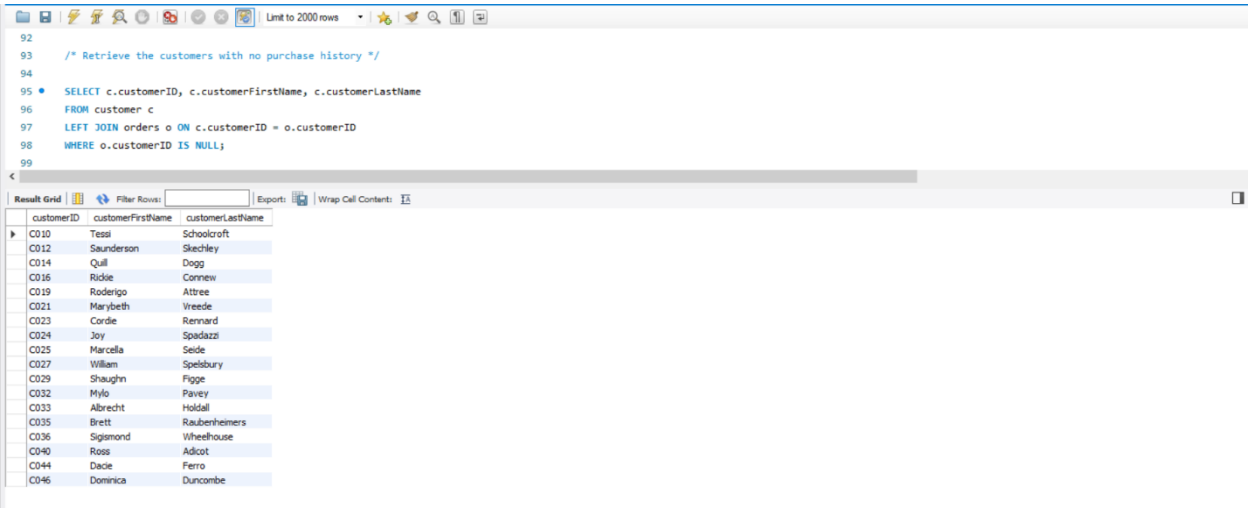
```
80
81    /* Retrieve the customers with the most purchase excluding customers with no orders */
82
83 •  SELECT customerID, customerFirstName, customerLastName, totalPurchaseAmount
84    FROM (
85        SELECT c.customerID, c.customerFirstName, c.customerLastName, SUM(o.orderPrice) AS totalPurchaseAmount
86        FROM customer c
87        INNER JOIN orders o ON c.customerID = o.customerID
88        GROUP BY c.customerID, c.customerFirstName, c.customerLastName
89    ) AS customerPurchases
90    ORDER BY totalPurchaseAmount DESC;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

customerID	customerFirstName	customerLastName	totalPurchaseAmount
C002	Ree	Paris	626.00
C030	Pierce	Novak	342.00
C004	Patsy	Caso	292.00
C042	Ron	Allridge	256.00
C026	Matilda	Frankema	215.00
C015	Willis	Kaesmakers	198.00
C018	Lauryn	Van Halen	181.00
C022	Jourdan	Heyfield	165.00
C011	Marielle	Ollerhead	151.00
C031	Donella	Orehead	151.00
C005	Cooper	Rawcliffe	150.00
C041	Zara	Bosomworth	150.00
C007	Murdoch	Falkner	149.00
C009	Anthia	Alf	130.00
C017	Brendan	Aves	126.00
C028	Griffith	Nutbeam	124.00
C001	Floris	Phelip	120.00
C047	Alric	Poyntz	113.00
C038	Fabien	Rzehorz	99.00

Query 6 Retrieves the customers with no purchase history.

Figure 11 Insights on No Purchase History Customers



```
/* Retrieve the customers with no purchase history */

SELECT c.customerID, c.customerFirstName, c.customerLastName
FROM customer c
LEFT JOIN orders o ON c.customerID = o.customerID
WHERE o.customerID IS NULL;
```

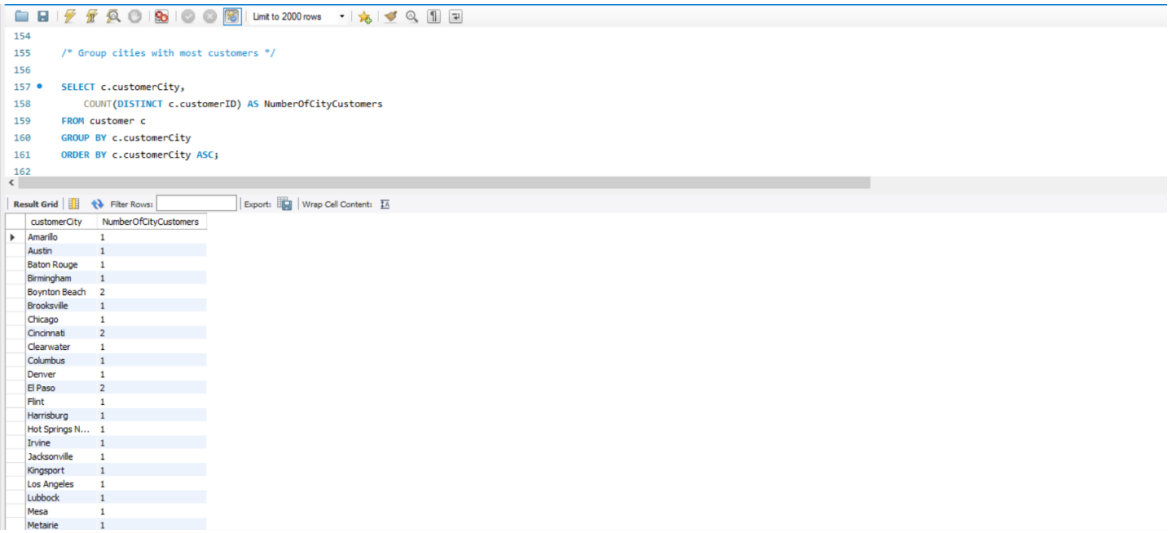
customerID	customerFirstName	customerLastName
C010	Tessi	Schoolcroft
C012	Saunderson	Skechley
C014	Quill	Dogg
C016	Rickie	Connew
C019	Roderigo	Attree
C021	Marybeth	Vreede
C023	Cordie	Rennard
C024	Joy	Spadazzi
C025	Marcella	Seide
C027	William	Spelsbury
C029	Shaughn	Figge
C032	Mylo	Pavey
C033	Albrecht	Holdall
C035	Brett	Raubenheimers
C036	Sigismond	Wheelhouse
C040	Ross	Adicot
C044	Dacie	Ferro
C046	Dominica	Duncombe

- I perform a LEFT JOIN between the "customer" table (aliased as "c") and the "orders" table (aliased as "o") on the "customerID" column.
- This ensures that all rows from the "customer" table are included, regardless of whether there are matching rows in the "orders" table.
- I then use a WHERE clause to filter out the rows where the "customerID" in the "orders" table is NULL, indicating that no orders have been placed by that customer.
- Finally, i select the "customerID" and "customerFirstName" and “customerLastName” columns from the "customer" table for the customers who haven't placed any orders.

This query will retrieve the customers who have not placed any orders.

Query 7 groups cities with most customers.

Figure 12 Insight on Customer Address



```
/* Group cities with most customers */

SELECT c.customerCity,
    COUNT(DISTINCT c.customerID) AS NumberOfCityCustomers
FROM customer c
GROUP BY c.customerCity
ORDER BY c.customerCity ASC;
```

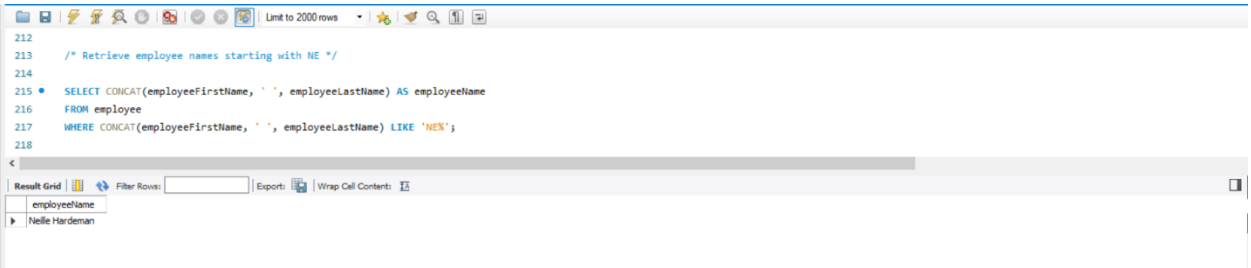
customerCity	NumberOfCityCustomers
Amarillo	1
Austin	1
Baton Rouge	1
Birmingham	1
Boynton Beach	2
Brooksville	1
Chicago	1
Cincinnati	2
Clearwater	1
Columbus	1
Denver	1
El Paso	2
Flint	1
Harrisburg	1
Hot Springs N...	1
Irvine	1
Jacksonville	1
Kingsport	1
Los Angeles	1
Lubbock	1
Mesa	1
Metairie	1

- I select the "customerCity", "customerID", "customerFirstName", and "customerLastName" columns from the "customer" table.
- The results are grouped by the "customerCity", "customerID", "customerFirstName", and "customerLastName" columns to group customers by identical cities and include detailed customer information.
- The COUNT(*) function calculates the number of customers for each city.
- Used ORDER BY clause at the end of the query.
- "c.customerCity ASC" orders the result set by the "customerCity" column in ascending alphabetical order.

This query will retrieve the total number of customers grouped by identical cities, along with their customer ID, first name, and last name. Adjust the column and table names as per your database schema.

Query 8 Retrieves employee names starting with NE.

Figure 13 Employee Name Characteristics



- `SELECT CONCAT(employeeFirstName, ' ', employeeLastName) AS employeeName`: Concatenates the `employeeFirstName` and `employeeLastName` columns with a space in between and aliases the result as `employeeName`.

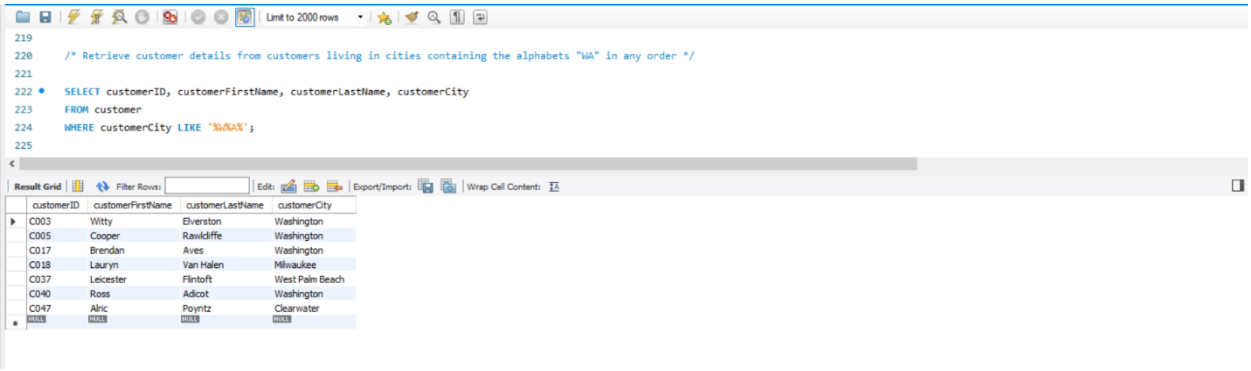
- `FROM employees`: Specifies the table to select the employee names from.

- `WHERE CONCAT(employeeFirstName, ' ', employeeLastName) LIKE 'NE%'`: Filters the results to include only those employee names that start with "NE". The `%` is a wildcard that matches any sequence of characters.

This query will return all employee names (concatenated from `employeeFirstName` and `employeeLastName`) starting with "NE".

Query 9 Retrieves customer details from customers living in cities containing the alphabets "WA" in any order.

Figure 14 Customer City Address



```
219
220    /* Retrieve customer details from customers living in cities containing the alphabets "WA" in any order */
221
222 •  SELECT customerID, customerFirstName, customerLastName, customerCity
223    FROM customer
224    WHERE customerCity LIKE '%W%A%';
225
```

Result Grid

customerID	customerFirstName	customerLastName	customerCity
C003	Witty	Elverston	Washington
C005	Cooper	Rawlcliffe	Washington
C017	Brendan	Aves	Washington
C018	Lauryn	Van Halen	Milwaukee
C037	Leicester	Flintoft	West Palm Beach
C040	Ross	Adicot	Washington
C047	Alric	Poyntz	Clearwater

- `SELECT customerFirstName, customerLastName, customerCity, customerID`: Specifies the columns you want to retrieve.
- `FROM customers`: Specifies the table to select the data from.
- `WHERE customerCity LIKE '%W%A%'`: Filters the results to include only those customers whose city contains the alphabets "W" and "A" in any order. The `%` is a wildcard that matches any sequence of characters.

This query will return the customer's first name, last name and ID of all customers living in a city with the alphabets "WA" in any order.

Views & Role

Views offer a convenient method to encapsulate complex SQL queries, acting as virtual tables accessible by name. I've created a view called "fullOrderDetails" to provide swift access to complete order details.

Figure 15 Views.

```
1  • use KicksFWDB;
2
3    /* create a view for the code that puts together all relevant columns from "employee" "customer" "product" "brand" "category" "orders" tables all together
4    to get indept insight on every order */
5
6  • CREATE VIEW fullOrderDetails AS
7    SELECT
8        orders.orderID,
9        orders.orderPrice,
10       orders.orderDate,
11       orders.orderStatus,
12       orders.productCode,
13       customer.customerID,
14       customer.customerFirstName,
15       customer.customerLastName,
16       product.price,
17       product.brandID,
18       brand.brandName,
19       Product.categoryType,
20       category.categoryName,
21       product.colour,
22       product.gender,
23       employee.employeeID,
24       employee.employeeFirstName,
25       employee.employeeLastName,
26       employee.employeePosition
27   FROM orders
28   JOIN employee ON orders.employeeID = employee.employeeID
29   JOIN customer ON orders.customerID = customer.customerID
30   JOIN product ON orders.productCode = product.productCode
```

Stored Procedure

Stored procedures at Kicks streamline crucial operations in the database. Three essential procedures are implemented: one for managing discounts on products priced at 100 or higher, another for adding new employees, and a third for removing records of ex-employees. See Appendix B for details.

Security & Backup

To safeguard sensitive data in the "employeePayroll" table, access has been restricted to authorized personnel – specifically, "HR Personnel," "Payroll Administrator," and "Finance Department" users. This strict measure ensures compliance with privacy regulations like the CCPA and bolsters data protection protocols.

Figure 16 Security & Backup

```
USE KicksFWDB;

/* dropping existing users */
DROP USER IF EXISTS 'HRPersonnel'@'localhost';
DROP USER IF EXISTS 'payrollAdministrator'@'localhost';
DROP USER IF EXISTS 'financeDepartment'@'local';

/* creating users */
CREATE USER 'HRPersonnel'@'localhost' IDENTIFIED BY 'HR123';
CREATE USER 'payrollAdministrator'@'localhost' IDENTIFIED BY 'PA123';
CREATE USER 'financeDepartment'@'local' IDENTIFIED BY 'FD123';

/* granting SELECT access to the "employeePayroll" table for the users */
GRANT SELECT ON KicksFWDB.employeePayroll TO 'HRPersonnel'@'localhost';
GRANT SELECT ON KicksFWDB.employeePayroll TO 'payrollAdministrator'@'localhost';
GRANT SELECT ON KicksFWDB.employeePayroll TO 'financeDepartment'@'local';
```

Performance Optimisation

Index optimisation

Figure 17 Performance.

```
Limit to 2000 rows
use KicksFWDB;

/* Add indexes to frequently queried columns */

CREATE INDEX idxCustomerID ON orders (customerID);
CREATE INDEX idxOrderDate ON orders (orderDate);
CREATE INDEX idxProductCode ON orders (productCode);
CREATE INDEX idxOrderStatus ON orders (orderStatus);
CREATE INDEX idxEmployeeID ON orders (employeeID);
CREATE INDEX idxPrice ON product (price);
CREATE INDEX idxCategoryType ON product (categoryType);
CREATE INDEX idxBrandID ON product (brandID);
CREATE INDEX idxCustomerFirstName ON customer (customerFirstName);
CREATE INDEX idxCustomerLastName ON customer (customerLastName);
CREATE INDEX idxEmployeeFirstName ON employee (employeeFirstName);
CREATE INDEX idxEmployeeLastName ON employee (employeeLastName);
```

This strategy focuses on optimizing database indexes to improve query performance. By creating additional indexes on columns frequently used in WHERE clauses or JOIN conditions, i aim to reduce query execution time by facilitating faster data retrieval.

Regular Maintenance

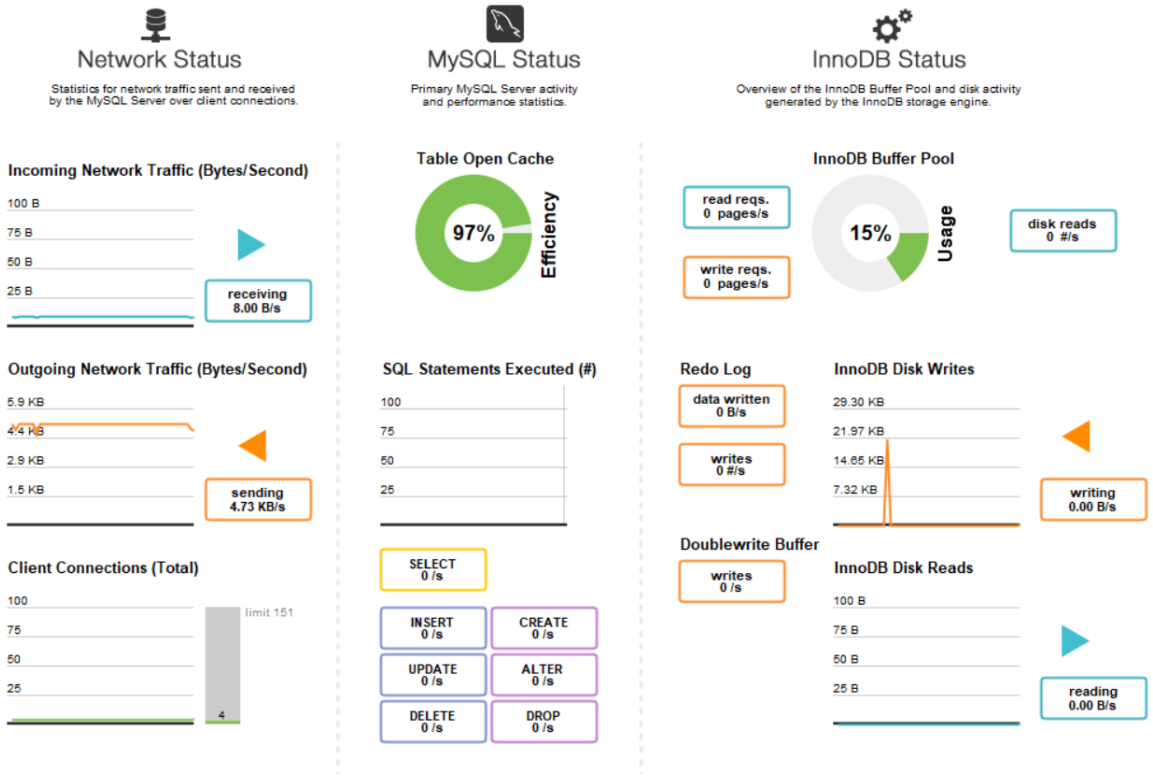
Figure 18 Maintenance.

```
19      /* Perform regular database maintenance tasks */
20 •    ANALYZE TABLE store;
21 •    ANALYZE TABLE employee;
22 •    ANALYZE TABLE employeepayroll;
23 •    ANALYZE TABLE customer;
24 •    ANALYZE TABLE brand;
25 •    ANALYZE TABLE category;
26 •    ANALYZE TABLE product;
27 •    ANALYZE TABLE orders;
28
```

Performing routine maintenance, such examining table statistics, is essential to maximising database performance. I help MySQL's query optimizer make better decisions while executing queries by making sure that table statistics are current, which eventually improves query performance over time.

Within MySQL Workbench, the Performance section offers a comprehensive dashboard for monitoring server performance. Presently, the performance metrics indicate satisfactory levels. However, ongoing vigilance is advised, especially with the anticipated growth of the database.

Figure 19 Performance Tab



Document Database Design

Logical Design

The initial design of a document database begins with capturing requirements, similar to the process outlined in the relational database, which leads to conceptual design. However, the subsequent stage adopts a query-driven approach to ascertain the optimal logical design.

Table 9 Document Table Relationships

Entity	Relationship	Entity	Description	Criticality
customer	One to Many	orders	Customers initiate purchases by placing orders.	Highly Critical Read
orders	One to Many	product	According to business rules, each order comprises a single product.	Highly Critical Write
orders	One to Many	employee	Employees are responsible for	Medium Write

			processing orders.	
store	One to Many	employee	The store hires employees.	Low Write

Considering the significance of orders, a straightforward approach would be to embed all attributes within the orders entity. However, this approach would result in large documents. A more sensible refinement would involve separating the entities for store, employees, and customers, and then embedding employees into stores for easier updates.

Figure 20 Document Logical Design

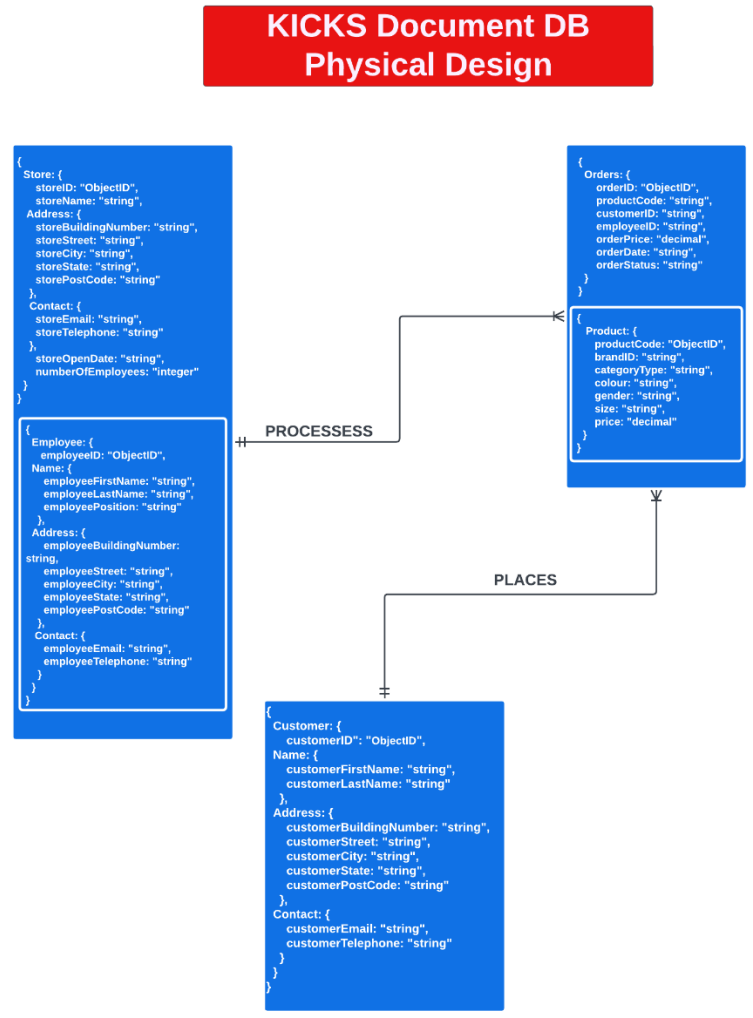
Physical Design

Finally, the variable types for each attribute were selected, documented, and presented in JSON format in the physical design.

Table 10 Variable Types

Variable	Rationale
ObjectID	Serves as the unique identifier for each document.
String	For fields that will contain alphanumeric characters or a combination of letters and numbers.
Integer	For fields that will contain only numerical values.
Decimal	For fields that require decimal precision.

Figure 21 Document Physical Design.



Conclusion

This project was undertaken to fulfill the outlined requirements and aid Kicks Ltd in establishing a user-friendly and readily accessible database for storing sales data. Design choices, implementation strategies, and query executions have been meticulously documented to ensure transparency and demonstrate the efficiency of the database. With potential expansion in mind, accommodating additional physical store locations, staff members, product offerings, and customer reach may lead to increased sales and data volume. This presents an opportunity for further database enhancement and functionality improvements to accommodate the growing needs of the business.

References

Amran, N., Mohamed, H., & Bahry, F. D. S. (2018). Developing Human Resource Training Management (HRTM) Conceptual Model Using Entity Relationship Diagram (ERD). *International Journal of Academic Research in Business and Social Sciences*, 8(12), pp. 1444–1459.

Connolly, T. and Begg, C. (2014) Database Systems: a Practical Approach to Design, Implementation, and Management. Global Edition. Harlow: Pearson Education, Limited.

Cvetkov-Iliev, A., Allauzen, A. & Varoquaux, G. (2023) 'Relational data embeddings for feature enrichment with background information', *Mach Learn* **112**, pp. 687–720 doi: https://doi.org/10.1007/s10994-022-06277-7.

L. M. Ferreira, S. N. Alves-Souza and L. M. Da Silva (2023). 'Multidimensional Modelling in NoSQL Database: A Systematic Review', *18th Iberian Conference on Information Systems and Technologies (CISTI)*, Aveiro, Portugal, 2023, pp. 1-6, doi: 10.23919/CISTI58278.2023.10211592.

Gartner .M (2023). 'Performance Benefits of NOT NULL Constraints on Foreign Key Reference Columns', Cockroach Labs. Available at: Performance Benefits of NOT NULL Constraints on Foreign Key Reference Columns (cockroachlabs.com)

Rajendran R.K., Priya T.M (2023). 'Designing an Efficient and Scalable Relational Database Schema: Principles of Design for Data Modeling', *The Software Principles of Design for Data Modeling*. Doi: 10.4018/978-1-6684-9809-5.ch013.

Saif Teria (2023) 'Introduction to Database Systems', *Algonquin College of Applied Arts and Technology Ottawa*, Ontario, Canada Edition 2.6-S.

Sahatqija, K. et al. (2018) 'Comparison between relational and NOSQL databases', *MIPRO 2018 - 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, Opatija, Croatia, 21-25 May 2018. Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8400041.

Sebastian Link, Henning Koehler, Aniruddh Gandhi, Sven Hartmann, Bernhard Thalheim. (2023). 'Cardinality constraints and functional dependencies in SQL: Taming data redundancy in logical database design', *Information Systems*, Volume 115. doi: https://doi.org/10.1016/j.is.2023.102208.

Appendices

Appendix A - Queries

Query 1 as shown in Figure 6. Retrieves all pending orders.

The SQL query employs the following statements:

“SELECT” Specifies the columns to be retrieved in the result set.
“(*)” The asterisk is a wildcard character that represents all columns in a table (It's a shorthand way of selecting all columns without explicitly listing them one by one).
“FROM” Specifies the table from which data will be retrieved.
“WHERE” Filters rows based on specified conditions.

This Query retrieves all the pending customer orders.

Query 2 as shown on Figure 7. Retrieves the top 5 best-selling brands in descending order.

- I start by using JOIN to join the "orders" table with the "product" table on the "productCode" column to link each order with its corresponding product.
- Then, i JOIN the "product" table with the "brand" table on the "brandID" column to associate each product with its brand.
- Next, i group the results by the brand name using the GROUP BY clause.
- I count the number of sales for each brand using the COUNT(*) function.
- Finally, i order the results by the total number of sales in descending order and limit the output to the top 5 brands using the LIMIT clause.

Furthermore, "b" refers to the alias assigned to the "brand" table. In SQL queries, aliases are used to provide shorter, more convenient names for tables or columns, making the query easier to read and write. In this case, "b" is used as an alias for the "brand" table, so when you see "b.brandName", it means the "brandName" column from the "brand" table. Alias’s remaing consistent where applicable throughout this report.

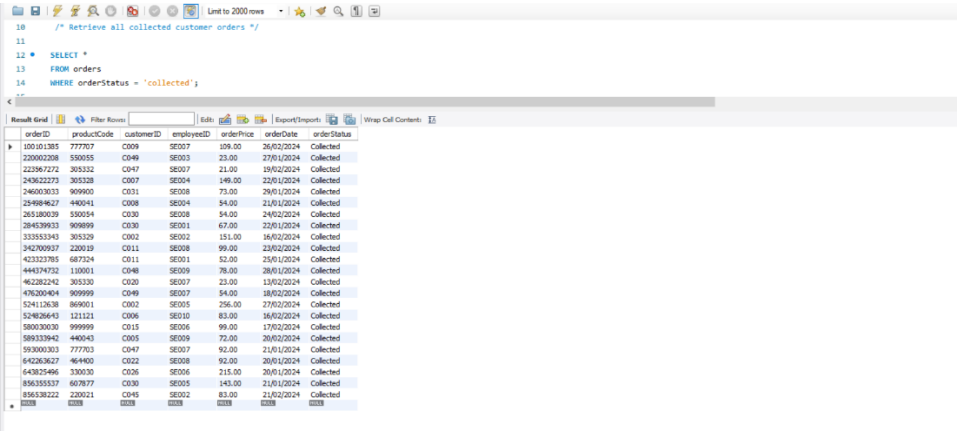
Query 4 as shown in Figure 10. Retrieves the customers with the most purchase excluding customers with no orders.

“INNER JOIN” the inner join in the provided code combines data from the "customer" and "orders" tables based on matching customer IDs. It selects only the rows where there is a match between the customer IDs in both tables, retrieving information about customers who have placed orders.

The explanation for this code focuses solely on the usage of the INNER JOIN statement, assuming familiarity with the other SQL statements employed in this query

Query 10 Retrieves all collected customer orders.

Figure 22 Insight on Collected Orders



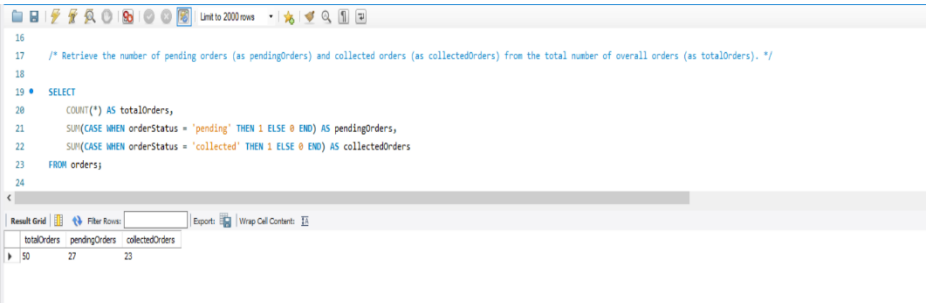
```
/* Retrieve all collected customer orders */

SELECT *
FROM orders
WHERE orderStatus = 'collected';
```

orderID	productCode	customerID	employeeID	orderPrice	orderDate	orderStatus
100101385	777707	C009	SE007	109.00	26/02/2024	Collected
220002208	550055	C049	SE003	23.00	27/01/2024	Collected
223367272	305332	C047	SE007	21.00	19/02/2024	Collected
243622273	305328	C007	SE004	149.00	22/01/2024	Collected
246003033	909900	C031	SE008	73.00	29/01/2024	Collected
254984627	440041	C008	SE004	54.00	21/01/2024	Collected
265180039	550054	C030	SE008	54.00	24/02/2024	Collected
284539933	909899	C030	SE001	67.00	22/01/2024	Collected
333553343	305329	C002	SE002	151.00	16/02/2024	Collected
342709937	220019	C011	SE008	99.00	23/02/2024	Collected
423323785	687324	C011	SE001	52.00	25/01/2024	Collected
444374732	110001	C048	SE009	78.00	28/01/2024	Collected
462282242	305330	C020	SE007	23.00	13/02/2024	Collected
476200404	909999	C049	SE007	54.00	18/02/2024	Collected
524112638	869001	C002	SE005	256.00	27/02/2024	Collected
524826643	121121	C006	SE010	83.00	16/02/2024	Collected
580030030	999999	C015	SE006	99.00	17/02/2024	Collected
589333942	440043	C005	SE009	72.00	20/02/2024	Collected
593000303	777703	C047	SE007	92.00	21/01/2024	Collected
642263627	464400	C022	SE008	92.00	20/01/2024	Collected
643825496	330030	C026	SE006	215.00	20/01/2024	Collected
856355537	607877	C030	SE005	143.00	21/01/2024	Collected
856538222	220021	C045	SE002	83.00	21/02/2024	Collected

This Query retrieves all the collected customer orders.

Query 11 Retrieves the number of pending orders (as pendingOrders) and collected orders (as collectedOrders) from the total number of overall orders (as totalOrders).

Figure 23 Overall Orders Description.



```
/* Retrieve the number of pending orders (as pendingOrders) and collected orders (as collectedOrders) from the total number of overall orders (as totalOrders). */

SELECT
    COUNT(*) AS totalOrders,
    SUM(CASE WHEN orderStatus = 'pending' THEN 1 ELSE 0 END) AS pendingOrders,
    SUM(CASE WHEN orderStatus = 'collected' THEN 1 ELSE 0 END) AS collectedOrders
FROM orders;
```

totalOrders	pendingOrders	collectedOrders
50	27	23

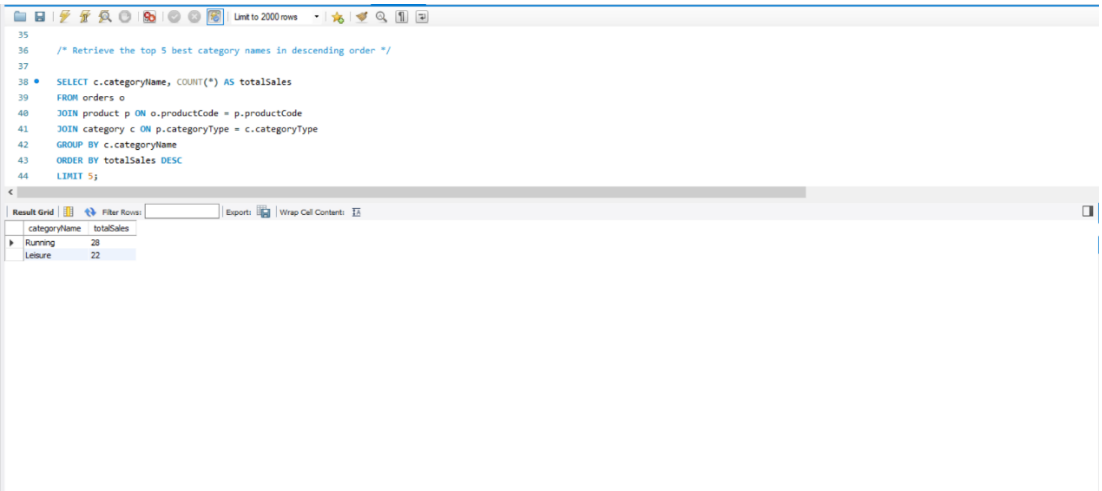
The SQL query employs various statements to analyze data from the "orders" table:

“SELECT” Specifies the columns to be retrieved in the result set.
“COUNT(*)” Counts the total number of rows in the table.
“SUM()” Calculates the sum of values meeting specified conditions.
“CASE WHEN” Evaluates conditions and returns a value based on the result.
“FROM” Specifies the table from which data will be retrieved.
“WHERE” Filters rows based on specified conditions.

This query will count the total number of orders (as totalOrders), the number of pending orders (as pendingOrders), and the number of collected orders (as collectedOrders). Representing them in a table.

Query 12 Retrieves the top 5 best category names in descending order.

Figure 24 Insight on Categories.

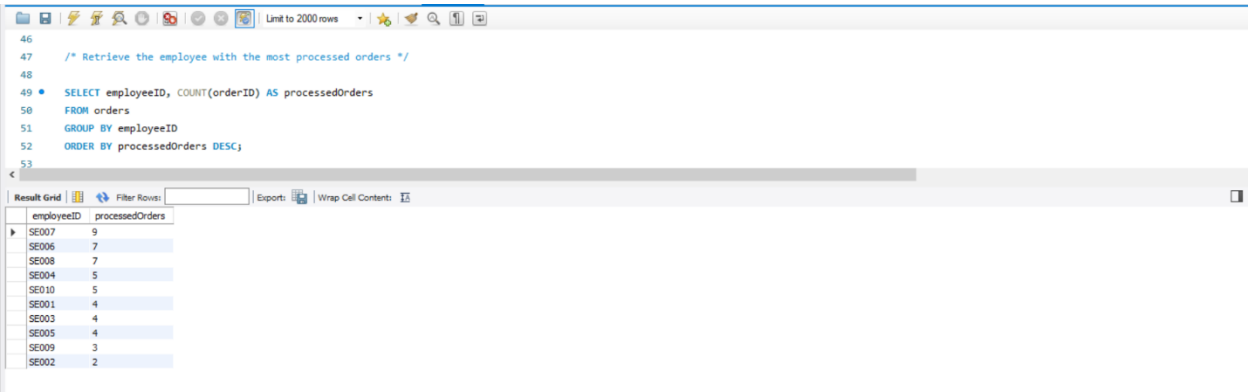


- I JOIN the "orders" table with the "product" table on the "productCode" column to link each order with its corresponding product.
- Then, i JOIN the "product" table with the "category" table on the "categoryType" column to associate each product with its category.
- Next, i group the results by the category name using the GROUP BY clause.
- I count the number of sales for each category using the COUNT(*) function.
- Finally, i order the results by the total number of sales in descending order and limit the output to the top 5 categories using the LIMIT clause.

This query will retrieve the top 5 selling category names based on the number of sales.

Query 13 Retrieves the employee with the most processed orders.

Figure 25 Employee Workrate



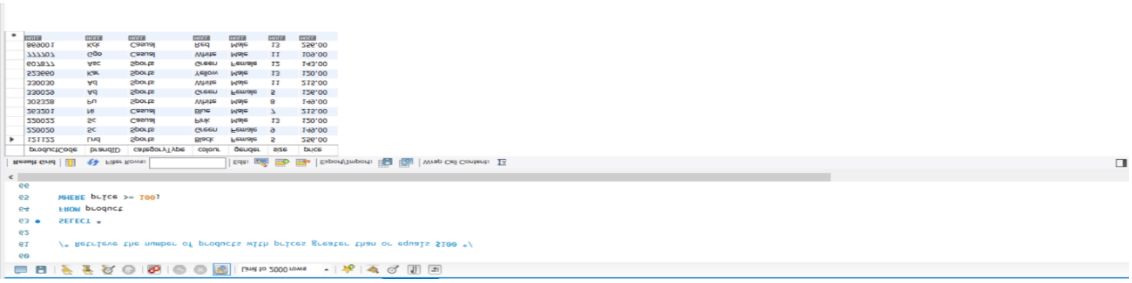
- I use the COUNT() function to count the number of orders processed by each employee.
- The GROUP BY clause groups the results by the "employeeID", so that the COUNT() function calculates the number of orders for each employee.

- I use the ORDER BY clause to sort the results in descending order based on the count of processed orders.

This query will retrieve the employee with the highest number of processed orders directly from the "orders" table.

Query 14 Retrieves the number of products with prices greater than or equals $100.

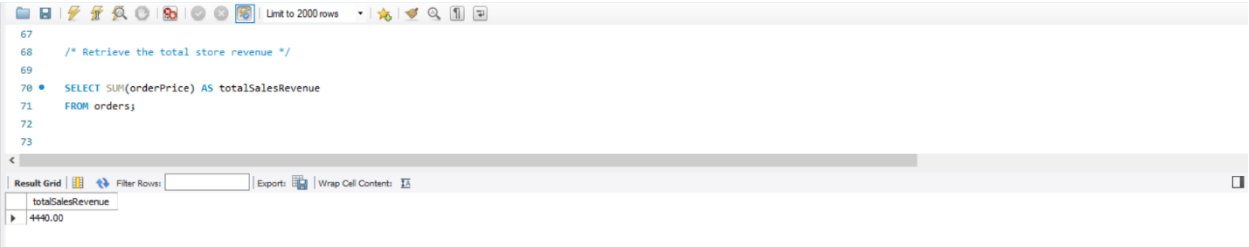
Figure 26 Expensive Products.



This query will retrieve all rows from the "products" table where the value in the "price" column is greater than or equals to 100 with the built in greater than or equals to function <=.

Query 15 Retrieves the total store revenue.

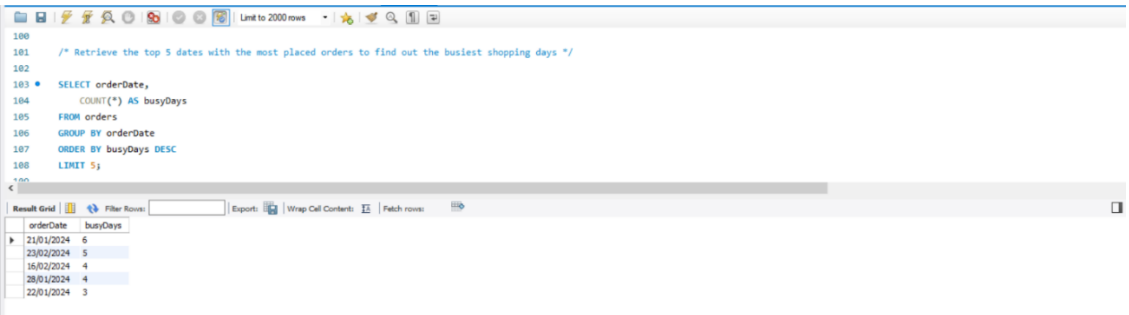
Figure 27 Insight on all Products.



This query will give you the total sales revenue from all orders in the "order" table.

Query 16 Retrieves the top 5 dates with the most placed orders to find out the busiest shopping days.

Figure 28 Sales Revenue

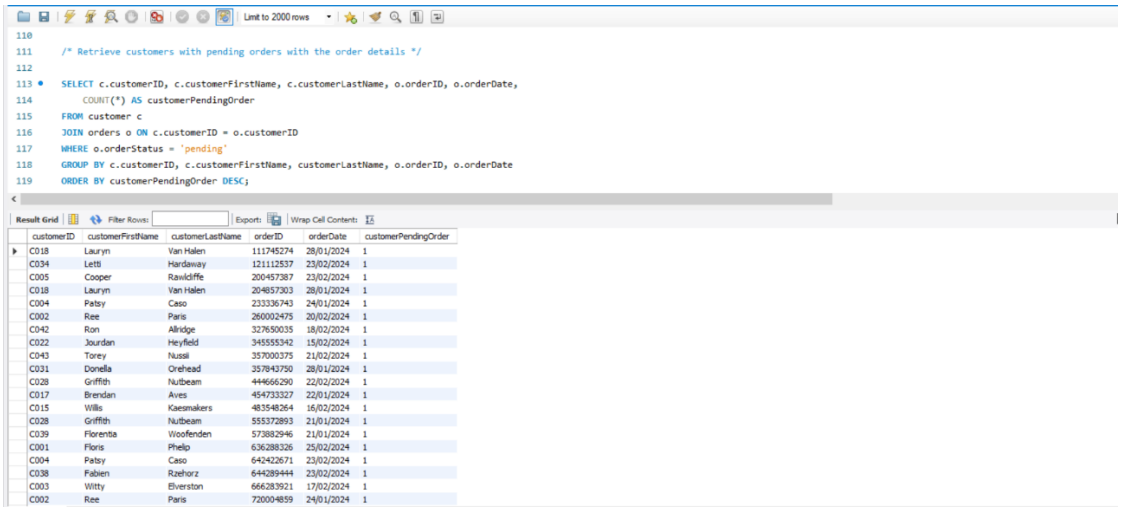


- I perform a LEFT JOIN between the "customer" table and the "orders" table on the "customerID" column.
- This ensures that all rows from the "customer" table are included, regardless of whether there are matching rows in the "orders" table.
- I then use a WHERE clause to filter out the rows where the "customerID" in the "orders" table is NULL, indicating that no orders have been placed by that customer.
- I use LIMIT 5 to limit the result to 5 in descending order.
- Finally, i select the "customerID" and "customerName" columns from the "customer" table for the customers who haven't placed any orders.

This query will retrieve the customers who have not placed any orders. Adjust the column and table names as per your database schema.

Query 17 Retrieves customers with pending orders with the order details.

Figure 29 Insights on Customers with Pending Orders



```
110
111    /* Retrieve customers with pending orders with the order details */
112
113 •  SELECT c.customerID, c.customerFirstName, c.customerLastName, o.orderID, o.orderDate,
114        COUNT(*) AS customerPendingOrder
115    FROM customer c
116    JOIN orders o ON c.customerID = o.customerID
117    WHERE o.orderStatus = 'pending'
118    GROUP BY c.customerID, c.customerFirstName, customerLastName, o.orderID, o.orderDate
119    ORDER BY customerPendingOrder DESC;
```

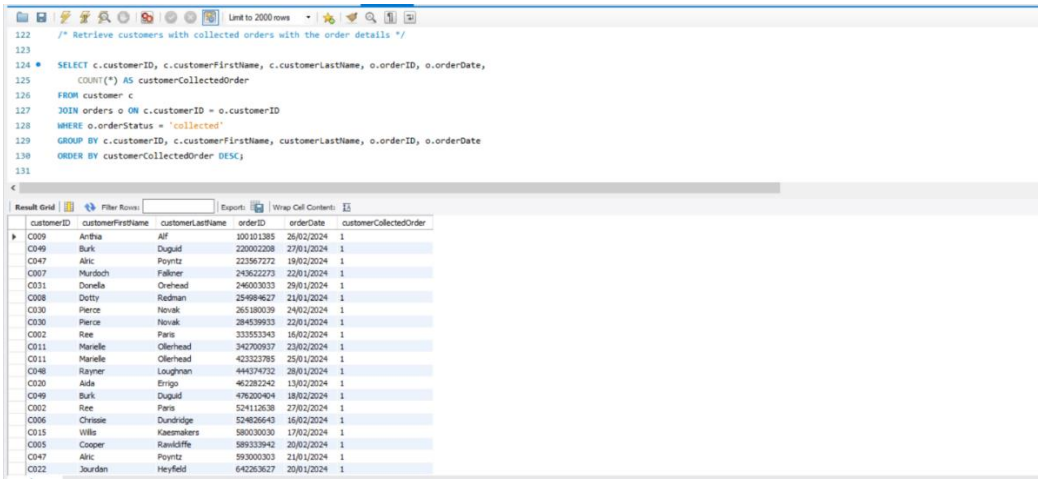
Result Grid

customerID	customerFirstName	customerLastName	orderID	orderDate	customerPendingOrder
C018	Lauryn	Van Halen	111745274	28/01/2024	1
C034	Letti	Hardaway	121112537	23/02/2024	1
C005	Cooper	Rawcliffe	200457387	23/02/2024	1
C018	Lauryn	Van Halen	204857303	28/01/2024	1
C004	Patsy	Caso	233336743	24/01/2024	1
C002	Ree	Paris	260002475	20/02/2024	1
C042	Ron	Allridge	327650035	18/02/2024	1
C022	Jourdan	Heyfield	345555342	15/02/2024	1
C043	Torey	Nussii	357000375	21/02/2024	1
C031	Donella	Orehead	357843750	28/01/2024	1
C028	Griffith	Nutbeam	444666290	22/02/2024	1
C017	Brendan	Aves	454733327	22/01/2024	1
C015	Willis	Kaesmakers	483548264	16/02/2024	1
C028	Griffith	Nutbeam	555372893	21/01/2024	1
C039	Florentia	Woofenden	573882946	21/01/2024	1
C001	Floris	Phelp	636288326	25/02/2024	1
C004	Patsy	Caso	642422671	23/02/2024	1
C038	Fabien	Rzehorz	644289444	23/02/2024	1
C003	Witty	Elverston	666283921	17/02/2024	1
C002	Ree	Paris	720004859	24/01/2024	1

- I join the "customer" table with the "orders" table on the "customerID", “orderID” and “orderDate” column.
- I filter for pending orders by adding a WHERE clause to only include rows where the orderStatus is 'pending'.
- The results are then grouped by customer using the GROUP BY clause.
- I use the COUNT(*) function to count the number of pending orders for each customer.
- Finally, i order the results by the count of pending orders in descending order using the ORDER BY clause.

Query 18 Retrieves customers with collected orders with the order details.

Figure 30 Customers with Collected Orders



```
/* Retrieve customers with collected orders with the order details */

SELECT c.customerID, c.customerFirstName, c.customerLastName, o.orderID, o.orderDate,
    COUNT(*) AS customerCollectedOrder
FROM customer c
JOIN orders o ON c.customerID = o.customerID
WHERE o.orderStatus = 'collected'
GROUP BY c.customerID, c.customerFirstName, customerLastName, o.orderID, o.orderDate
ORDER BY customerCollectedOrder DESC;
```

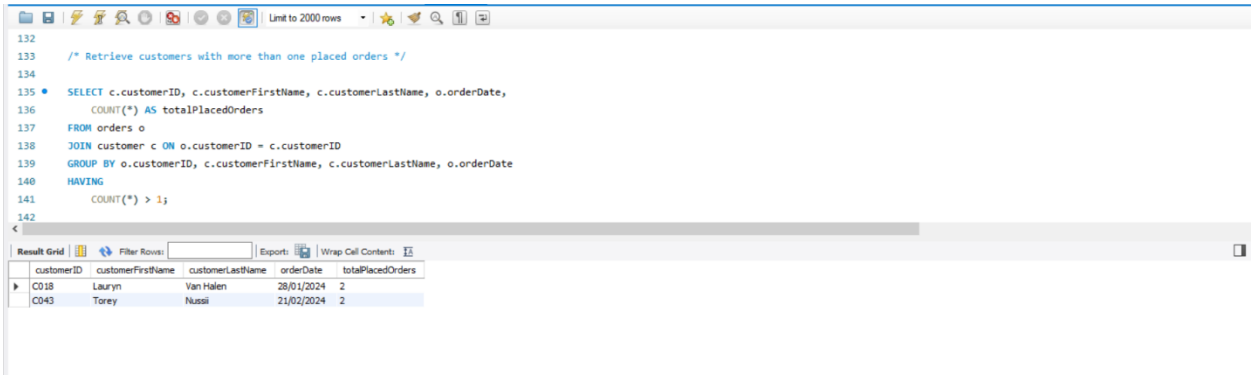
customerID	customerFirstName	customerLastName	orderID	orderDate	customerCollectedOrder
C009	Anthia	Alf	100101385	26/02/2024	1
C049	Burk	Duguid	220002208	27/01/2024	1
C047	Alric	Poyntz	223567272	19/02/2024	1
C007	Murdoch	Falkner	243622273	22/01/2024	1
C031	Donella	Orehead	246003033	29/01/2024	1
C008	Dotty	Redman	254984627	21/01/2024	1
C030	Pierce	Novak	265180039	24/02/2024	1
C030	Pierce	Novak	284539933	22/01/2024	1
C002	Ree	Paris	333553343	16/02/2024	1
C011	Marielle	Ollerhead	342700937	23/02/2024	1
C011	Marielle	Ollerhead	423323785	25/01/2024	1
C048	Rayner	Loughnan	444374732	28/01/2024	1
C020	Aida	Errigo	462282242	13/02/2024	1
C049	Burk	Duguid	476200404	18/02/2024	1
C002	Ree	Paris	524112638	27/02/2024	1
C006	Chrissie	Dundridge	524826643	16/02/2024	1
C015	Willis	Kaesmakers	580030030	17/02/2024	1
C005	Cooper	Rawcliffe	589333942	20/02/2024	1
C047	Alric	Poyntz	593000303	21/01/2024	1
C022	Jourdan	Heyfield	642263627	20/01/2024	1

In this query: - I am selecting customer details (customerID and customerName), along with order details (orderID and orderDate).
- I'm filtering for collected orders by specifying `WHERE o.orderStatus = 'collected'`.
- The results are grouped by customer and order details to count the number of collected orders for each customer.
- Finally, the results are ordered by the count of collected orders in descending order.

This query will provide detailed information about each collected order, including the customer details, order ID, order date, and the count of collected orders.

Query 19 Retrieves customers with more than one placed order.

Figure 31 Insight on Placed Orders



```
/* Retrieve customers with more than one placed orders */

SELECT c.customerID, c.customerFirstName, c.customerLastName, o.orderDate,
    COUNT(*) AS totalPlacedOrders
FROM orders o
JOIN customer c ON o.customerID = c.customerID
GROUP BY o.customerID, c.customerFirstName, c.customerLastName, o.orderDate
HAVING
    COUNT(*) > 1;
```

customerID	customerFirstName	customerLastName	orderDate	totalPlacedOrders
C018	Lauryn	Van Halen	28/01/2024	2
C043	Torey	Nussii	21/02/2024	2

- I select the "customerID", "customerFirstName", "customerLastName", and "orderDate" columns from the "customer" and "orders" tables.

- The JOIN clause is used to join the "orders" table with the "customer" table based on the "customerID" column.

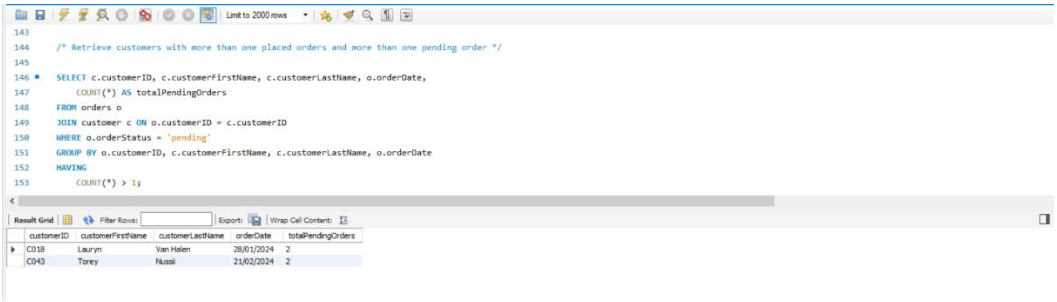
- The results are grouped by the "customerID", "customerFirstName", "customerLastName", and "orderDate" columns to count the number of orders for each customer.

- The HAVING clause filters the groups to include only those with a count of orders greater than 1, indicating customers who have placed more than one order.

This query will retrieve customers who have placed more than one order, along with their first name, last name, order date, and the total number of orders placed.

Query 20 Retrieves customers with more than one placed orders and more than one pending order.

Figure 32 Multiple Transaction Customer



- I select the "customerID", "customerFirstName", "customerLastName", and "orderDate" columns from the "customer" and "order" tables.

- The JOIN clause is used to join the "orders" table with the "customer" table based on the "customerID" column.

- The WHERE clause filters for collected orders by specifying `o.orderStatus = “pending”.

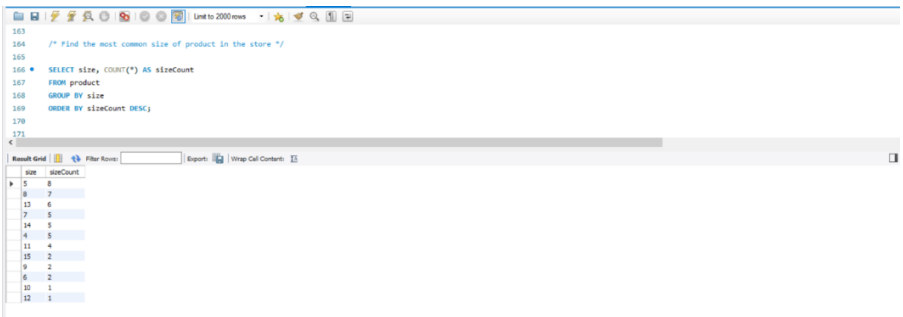
- The results are grouped by the "customerID", "customerFirstName", "customerLastName", and "orderDate" columns to count the number of pending orders for each customer.

- The HAVING clause filters the groups to include only those with a count of pending orders greater than 1, indicating customers who have more than one pending order.

This query will retrieve customers who have more than one pending order, along with their first name, last name, order date, and the total number of pending orders placed.

Query 21 Finds the most common size of product in the store.

Figure 33 Stock Check



- I select the "size" column from the "product" table.

- The results are grouped by the "size" column to count the occurrence of each size.

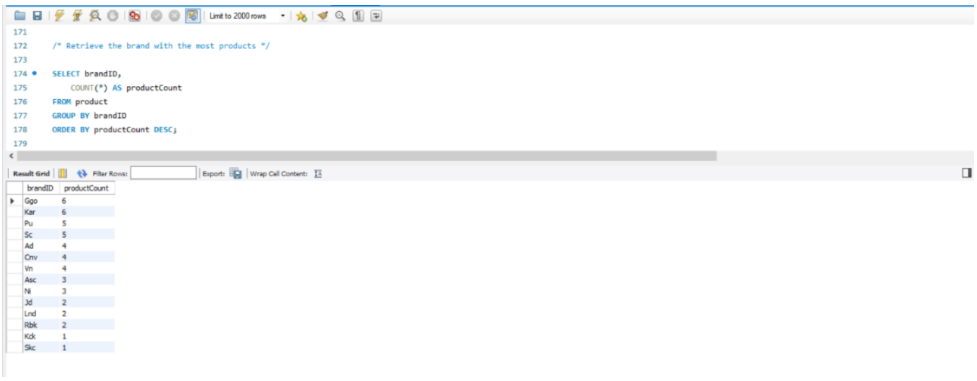
- The COUNT(*) function calculates the number of products for each size.

- I order the results by the count of products in descending order to find the most common size.

This query will retrieve the most common size of products in the store along with the count of products for that size. Adjust the column and table names as per your database schema.

Query 22 Retrieves the brand with the most products.

Figure 34 Brand Supply



- I select the "brandID" column from the "product" table.

- The results are grouped by the "brandID" column to count the number of products for each brand.

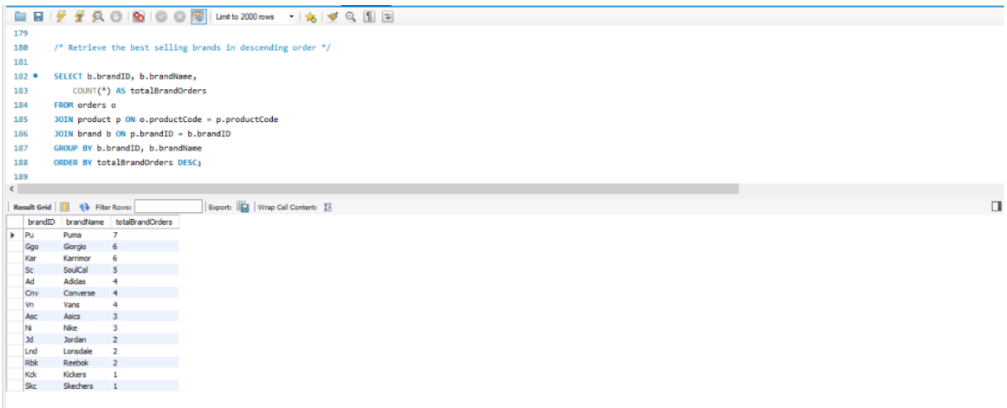
- The COUNT(*) function calculates the number of products for each brand.

- I order the results by the count of products in descending order to find the brand with the most products.

This query will retrieve the brand ID with the most products.

Query 23 Retrieves the best-selling brands in descending order.

Figure 35 Best Selling Brands



- I select the "brandID" and "brandName" columns from the "brand" table.

- I join the "orders" table with the "product" table on the "productCode" column to associate products with their brands.

- I then join the "product" table with the "brand" table on the "brandID" column to get the brand details.

- The results are grouped by the "brandID" and "brandName" columns to count the number of orders for each brand.

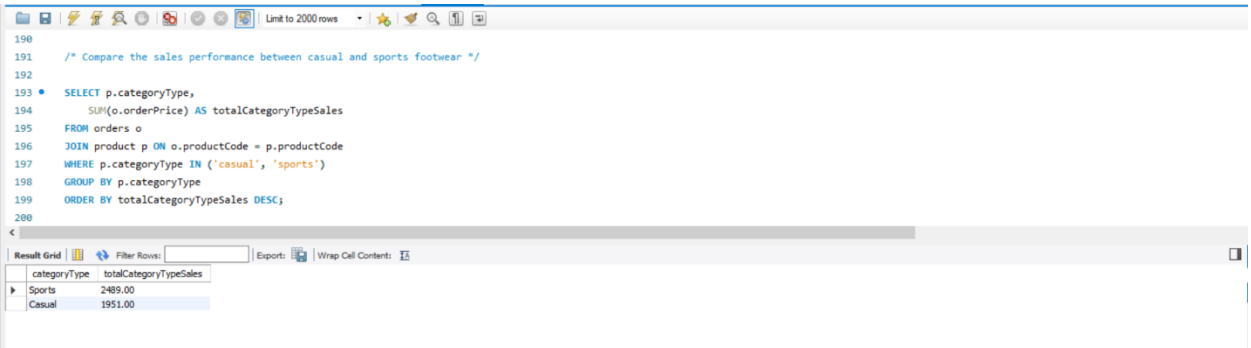
- The COUNT(*) function calculates the number of orders for each brand.

- I order the results by the totalBrandOrders in descending order to find the highest selling brands.

This query will retrieve the brand ID, brand name, and the totalBrandOrders for each brand, sorted by the totalBrandOrders in descending order.

Query 24 Compares the sales performance between casual and sports footwear.

Figure 36 Comparison between Categories.



- I select the "categoryType" column from the "product" table to represent the shoe categories.

- I sum the "orderPrice" from the "orders" table to calculate the total sales for each category.

- I join the "orders" table with the "product" table using the "productCode" column.

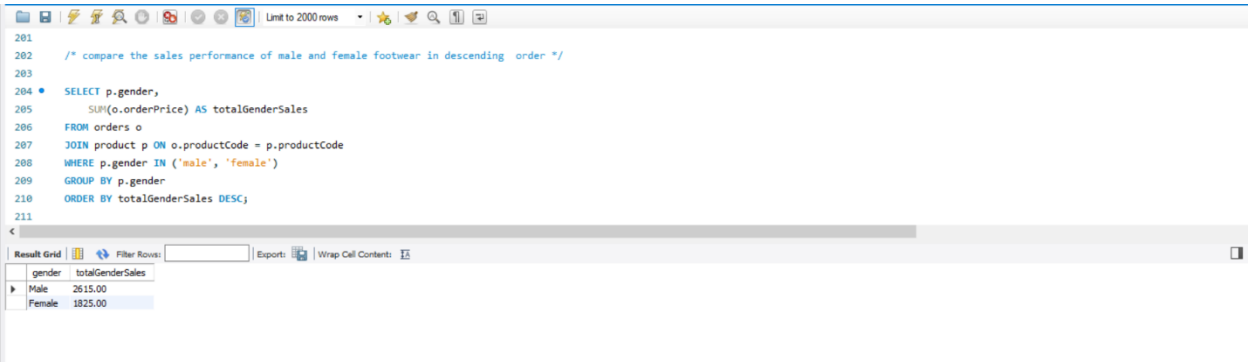
- I filter the orders based on the product category, including only 'casual' and 'sports'.

- The results are grouped by the "categoryType" column to show the total sales for each category.

This query will retrieve the total sales for casual and sports shoes separately to make possible comparison of the total sales for each category to analyze the sales performance between casual and sports shoes.

Query 25 compares the sales performance of male and female footwear in descending order.

Figure 37 Gender Sales



```
/* compare the sales performance of male and female footwear in descending  order */

SELECT p.gender,
    SUM(o.orderPrice) AS totalGenderSales
FROM orders o
JOIN product p ON o.productCode = p.productCode
WHERE p.gender IN ('male', 'female')
GROUP BY p.gender
ORDER BY totalGenderSales DESC;
```

gender	totalGenderSales
Male	2615.00
Female	1825.00

- I select the "gender" column from the "product" table to represent the gender categories.

- I sum the "orderPrice" from the "orders" table to calculate the total sales for each gender category.

- I join the "orders" table with the "product" table using the "productCode" column.

- I filter the orders based on the gender category, including only 'male' and 'female'.

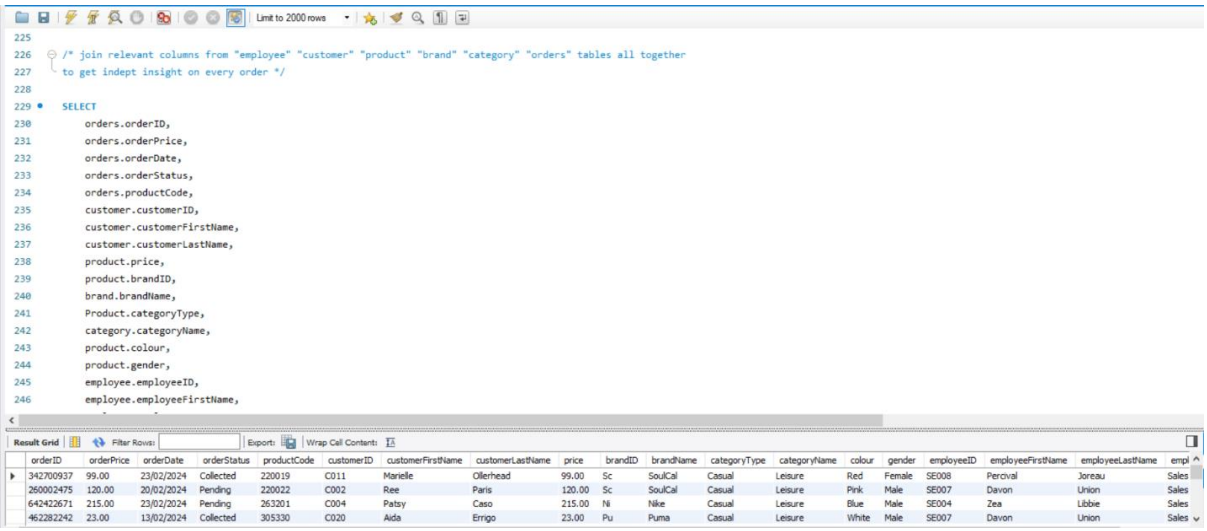
- The results are grouped by the "gender" column to show the totalGenderSales for each gender category.

- I order the results by the totalGenderSales in descending order to display the gender category with the highest sales first.

This query will retrieve the total sales for male and female shoes separately, with the category having the highest sales appearing first.

Query 26 joins relevant columns from "employee" "customer" "product" "brand" "category" "orders" tables all together to get indept insight on every order.

Figure 38 Table Join.



```
/* join relevant columns from "employee" "customer" "product" "brand" "category" "orders" tables all together
to get indept insight on every order */

SELECT
    orders.orderID,
    orders.orderPrice,
    orders.orderDate,
    orders.orderStatus,
    orders.productCode,
    customer.customerID,
    customer.customerFirstName,
    customer.customerLastName,
    product.price,
    product.brandID,
    brand.brandName,
    Product.categoryType,
    category.categoryName,
    product.colour,
    product.gender,
    employee.employeeID,
    employee.employeeFirstName,
```

orderID	orderPrice	orderDate	orderStatus	productCode	customerID	customerFirstName	customerLastName	price	brandID	brandName	categoryType	categoryName	colour	gender	employeeID	employeeFirstName	employeeLastName	empl
342700937	99.00	23/02/2024	Collected	220019	C011	Marielle	Ollerhead	99.00	Sc	SoulCal	Casual	Leisure	Red	Female	SE008	Percival	Joreau	Sales
260002475	120.00	20/02/2024	Pending	220022	C002	Ree	Paris	120.00	Sc	SoulCal	Casual	Leisure	Pink	Male	SE007	Davon	Union	Sales
642422671	215.00	23/02/2024	Pending	263201	C004	Patsy	Caso	215.00	Ni	Nike	Casual	Leisure	Blue	Male	SE004	Zea	Libbie	Sales
462282242	23.00	13/02/2024	Collected	305330	C020	Aida	Errigo	23.00	Pu	Puma	Casual	Leisure	White	Male	SE007	Davon	Union	Sales

This query will provide the full details of every order, including details of the customer who placed the order, the employee who processed the order, and the product, brand, and category associated with the order.

Appendix B - Stored Procedure

stored procedure to retrieve all products with a price equal to or greater than $100.

Figure 39 Stored Procedures.

```
/* Drop existing stored procedures if they exist */

DROP PROCEDURE IF EXISTS GetProductsGreaterThan100;
DROP PROCEDURE IF EXISTS AddEmployee;
DROP PROCEDURE IF EXISTS RemoveEmployee;

/* stored procedure to retrieve all products with a price equal to or greater than 100 */

DELIMITER //
CREATE PROCEDURE GetProductsGreaterThan100()
BEGIN
    SELECT * FROM product WHERE price >= 100;
END //
DELIMITER ;


/* execute stored procedure and retrieve all products with a price equal to or greater than 100 */

CALL GetProductsGreaterThan100();
```

stored procedure for adding employee.

Figure 40 Adding Employees.

```
/* stored procedure for adding employee */

DELIMITER //
CREATE PROCEDURE AddEmployee(
    IN empID VARCHAR(55),
    IN firstName VARCHAR(55),
    IN lastName VARCHAR(55),
    IN position VARCHAR(55),
    IN buildingNumber VARCHAR(55),
    IN street VARCHAR(55),
    IN city VARCHAR(55),
    IN state VARCHAR(55),
    IN postCode VARCHAR(55),
    IN email VARCHAR(55),
    IN telephone VARCHAR(55),
    IN NI VARCHAR(255),
    IN startDate VARCHAR(55),
    IN endDate VARCHAR(55),
    IN taxCode VARCHAR(55),
    IN salaryAmt VARCHAR(55)
)
BEGIN
    -- Insert into employee table
    INSERT INTO employee (employeeID, employeeFirstName, employeeLastName, employeePosition, employeeBuildingNumber,
    employeeStreet, employeeCity, employeeState, employeePostCode, employeeEmail, employeeTelephone)
    VALUES (empID, firstName, lastName, position, buildingNumber,
    street, city, state, postCode, email, telephone);

    -- Insert into employeePayroll table
    INSERT INTO employeePayroll (NI, employeeID, employeeStartDate, employeeEndDate, taxCode, salary)
    VALUES (NI, empID, startDate, endDate, taxCode, salaryAmt);
END //
DELIMITER ;

/* execute stored procedure for adding employee */
CALL AddEmployee('SE011', 'Lucky', 'Ankara', 'Sales Executive', '123', 'Woji', 'Sacramento', 'California', '94427',
                 'luckyankara@kicks.com', '916-827-2947', '433-92-7468', '15/03/2024', '31/12/2024', '52836-0002', '18000');
```

As shown in Figure 40, the "CALL AddEmployee" statement includes the details of the new employee being added. Upon execution, Table 11 confirms that Lucky Ankara has been successfully added to the Kicks workforce.Execute stored procedure for adding employee = CALL AddEmployee()

Table 11 Confirmation of Added New Employee.

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

employeeID	employeeFirstName	employeeLastName	employeePosition	employeeBuildingNumber	employeeStreet	employeeCity	employeeState	employeePostCode	employeeEmail	employeeTelephone
M001	Mathias	Neylan	Manager	7	Ridgeview	New Orleans	Louisiana	70187	mneylan0@kicks.com	504-488-9812
SE001	Neille	Hardeman	Sales Executive	691	Vermont	Spokane	Washington	99210	nhardeman1@kicks.com	509-785-6448
SE002	Nappie	Dixey	Sales Executive	180	Jackson	Springfield	Ohio	45505	ndixey2@kicks.com	937-513-7097
SE003	Cecily	Sherbrook	Sales Executive	4	Sachs	Salt Lake City	Utah	84130	csherbrook3@kicks.com	801-939-4254
SE004	Zea	Libbie	Sales Executive	96	Village Green	New York City	New York	10105	zlibbie4@kicks.com	917-505-3894
SE005	Svend	Kirkbride	Sales Executive	98	Heath	Gainesville	Georgia	30506	skirkbride5@kicks.com	770-165-2732
SE006	Janos	Dourin	Sales Executive	60	Sunnyside	Sacramento	California	94263	jdourin6@kicks.com	916-942-1016
SE007	Davon	Union	Sales Executive	14	Parkside	Cincinnati	Ohio	45249	dunion7@kicks.com	513-852-2409
SE008	Percival	Joreau	Sales Executive	71	Milwaukee	Salem	Oregon	97306	pjoreau8@kicks.com	503-455-4184
SE009	Marielle	Carrane	Sales Executive	9	Merrick	Seattle	Washington	98195	mcarrane9@kicks.com	206-374-1063
SE010	Cesaro	Kuhwald	Sales Executive	21	Norway Maple	Flushing	New York	11355	ckuhwalda@kicks.com	917-860-4295
SE011	Lucky	Ankara	Sales Executive	123	Woji	Sacramento	California	94427	luckyankara@kicks.com	916-827-2947

Stored procedures are saved in the "Stored Procedure" section of the workbench, allowing for easier execution of complex database operations, such as adding a new employee, without the need to repeat lengthy code. They offer benefits like improved performance, enhanced security, modularity, ease of maintenance, and reduced network traffic.

stored procedure for removing employee.

Figure 41 Removing employes.

```
53
54      /* stored procedure for removing employee */
55
56      DELIMITER //
57      CREATE PROCEDURE RemoveEmployee(
58          IN employeeID  VARCHAR(55)
59      )
60      BEGIN
61          -- Delete from employee table
62          DELETE FROM employee WHERE employeeID = employeeID;
63
64          -- Delete from employeePayroll table
65          DELETE FROM employeePayroll WHERE employeeID = employeeID;
66      END //
67      DELIMITER ;
68
```

Execute stored procedure for removing employee = CALL RemoveEmployee()

Appendix C – Triggers

Triggers in database systems are specialized stored procedures activated by specific events like INSERT, UPDATE, or DELETE on a table. They can execute either before (BEFORE) or after (AFTER) the event and are crucial for:

Data Validation: Ensuring data integrity by validating entries.

Audit Logging: Logging changes for auditing purposes.

Automated Actions: Triggering actions like notifications or updates based on database events.

However, while powerful, poorly designed triggers can impact database performance, especially with high-frequency events.

For Kicks, I've set up a trigger that automatically dispatches a "Welcome to Kicks" email to new employees when they are added to the workforce. Additionally, I've created a stored procedure to facilitate the sending of this welcome email.

Figure 42 Email Trigger Upon Addition of Employee

```
use KicksFWDB;

/* Drop Existing send_welcome_email_after_insert Trigger */
DROP TRIGGER IF EXISTS send_welcome_email_after_insert;

/* Drop Existing SendWelcomeEmail Procedure */
DROP PROCEDURE IF EXISTS SendWelcomeEmail;

/* Create a trigger that automatically sends a welcome email when a new employee is added */
DELIMITER //
CREATE TRIGGER send_welcome_email_after_insert
AFTER INSERT ON employee
FOR EACH ROW
BEGIN
    -- Execute stored procedure to send welcome email
    CALL SendWelcomeEmail(NEW.employeeEmail);
END //
DELIMITER ;
```

Figure 43 Email Stored Procedure Associated with Email Trigger

```

/* Create Stored Procedure that automatically sends a welcome email when a new employee */
DELIMITER //
CREATE PROCEDURE SendWelcomeEmail(IN emailAddress VARCHAR(255))
BEGIN
    DECLARE emailSubject VARCHAR(255);
    DECLARE emailBody VARCHAR(1000);

    -- Define email subject and body
    SET emailSubject = 'Welcome to Kicks';
    SET emailBody = 'Dear Employee, Welcome to Kicks! We are excited to have you on board.';

    CALL YourEmailSendingFunction(emailAddress, emailSubject, emailBody);
END //
DELIMITER ;

```

Appendix D – Back Ups and Dump

Backing up and dumping data are crucial processes in database management.

Backup: Backing up a database involves creating a copy of the entire database, including all its tables, data, and configurations. This copy serves as a safeguard against data loss due to various reasons like system failures, hardware issues, or human errors. In essence, a backup allows you to restore your database to its previous state if any unexpected issues arise.

Dump: Dumping a database refers to extracting and saving its data in a specific format, typically a text file containing SQL statements. This dump file can then be used to recreate the database or transfer the database to another system. Dumping is particularly useful for migrating data between different database management systems or versions.

In summary, while a backup is a comprehensive copy of the entire database aimed at data protection, a dump is a formatted extract of the database that can be used for various purposes, including migration and restoration. Both processes are essential for ensuring data integrity and availability. I have successfully performed a backup and dump process for the Kicks database in MySQL, securing the company's data and enabling seamless migration or restoration as needed.