

Advanced Database Management System Design and Performance Analysis for E-commerce

Team Members:

- ☐ Alekhya Thalla
- ☐ Alka Santosh Naik
- ☐ Niveditha Yeginati
- ☐ Hima Karan Pusarla
- ☐ Raajitha Sai Bondada
- ☐ Koushik Reddy Gadipalli

EXECUTIVE SUMMARY

This project delivers a refined e-commerce database system, emphasizing performance optimization. Leveraging "Online Retail" and "Customer Demographics" datasets, we engineered a relational database schema detailed through an Entity-Relationship Diagram (ERD), ensuring integrity and efficiency.

Performance Tuning:

Critical performance enhancements were achieved through targeted experiments on indexing and optimizer settings. Indexing strategies significantly reduced query times, affirming their importance in operational efficiency. Optimizer adjustments further halved the execution times of complex queries, demonstrating the effectiveness of fine-tuning database configurations.

Insights and Analytics:

The system's capability was showcased by developing queries that provide business intelligence, such as sales trends and customer spending patterns—insights crucial for informed decision-making.

Conclusion:

The project exemplifies the integration of strategic database design with performance tuning to meet the demands of modern e-commerce businesses. The final system showcases a scalable, reliable database with enhanced performance, ready to support data-driven decision-making in the e-commerce sector.

Topic Area	Description	Points
Database Design	This part should include a logical database design (for the relational model), using normalization to control redundancy and integrity constraints for data quality.	Default: 25 Range: 20 - 30
Query Writing	This part is another chance to write SQL queries, explore transactions, and even do some database programming for stored procedures.	Default: 25 Range: 20 - 30
Performance Tuning	In this section, you can capitalize and extend your prior experiments with indexing, optimizer modes, partitioning, parallel execution and any other techniques you want to further explore.	Default: 25 Range: 20 - 30
Other Topics	Here you are free to explore any other topics of interest. Suggestions include DBA scripts, database security, interface design, data visualization, data mining, and NoSQL databases.	Default: 25 Range: 10 - 40

Dataset 1: Online Retail

Source: Online Retail Dataset

Data Description:

The Online Retail dataset captures transactions that occurred between 01/12/2010 and 09/12/2011 for a UK-based non-store online retail company. This company specializes in selling unique all-occasion gifts, catering mainly to wholesalers.

Dataset Characteristics:

Multivariate: The dataset contains multiple variables, enabling a comprehensive analysis of various aspects.

Sequential: Data is organized sequentially, likely based on transaction timestamps, providing insights into the chronological order of transactions.

Time-Series: As it spans transactions over a period, the dataset qualifies as time-series data, allowing us to explore trends and patterns over time.

Dataset Size:

Number of Instances: 1,00,000

Number of Features: 6

Features:

InvoiceNo: A unique 6-digit integral number categorizing each transaction. Transactions beginning with 'c' denote cancellations.

StockCode: A 5-digit integral number identifying each distinct product.

Description: Product names, represented as strings.

Quantity: Integer values indicating the quantities of each product per transaction.

InvoiceDate: Date and time when transactions were generated.

UnitPrice: Continuous feature denoting the product's price per unit in sterling.

CustomerID: A unique 5-digit integral number assigned to each customer.

Country: Country names where customers reside.

Additional Information:

The dataset is devoid of missing values, ensuring data completeness and reliability.

By leveraging this dataset, we aim to enrich our database system with valuable insights into customer behavior, product preferences, and transaction patterns. This information will empower us to optimize our e-commerce platform and enhance customer satisfaction.

Dataset 2: Customer Demographic and Behavioral Data

Source: Customer Demographic Dataset

Data Description:

This dataset provides comprehensive insights into customers' demographics and purchasing behavior. Specifically designed for market segmentation, it delves into purchasing patterns, aiding in the understanding of customer traits and the customization of targeted marketing strategies. The dataset encompasses key aspects of customer profiles, including gender, age, income, and a derived "Spending Score," reflecting customer purchasing behavior and frequency.

Dataset Characteristics:

Univariate & Multivariate: The dataset offers multiple variables for each customer, allowing both individual and combined analyses.

Cross-Sectional: Representing a static snapshot, the data captures customer information at a specific point in time.

Associative: Enables analysis of relationships between different variables, such as age and spending score, providing valuable insights into customer behavior.

Subject Area: Marketing, Retail, Consumer Behavior

Dataset Size:

Number of Instances: 200

Number of Features: 5 (excluding CustomerID)

Features

CustomerID: A unique categorical identifier for each customer, facilitating individual tracking.

Genre: A categorical variable denoting the gender of the customer.

Age: A numerical variable representing the age of the customer, offering insights into age-related preferences and behaviors.

Annual_Income_(k\$): A numerical variable indicating the customer's annual income in thousands of dollars. This critical variable provides key insights into purchasing power and consumer affordability.

Spending_Score: A numerical score assigned to customers based on their purchasing patterns. This constructed variable offers a concise representation of customer behavior, likely formulated through proprietary algorithms or specific business rules.

Additional Information

The dataset is complete, devoid of any missing values, ensuring data integrity and reliability for analysis. By leveraging this dataset, we aim to enhance our understanding of customer behavior and preferences, enabling us to tailor our services and marketing efforts to specific customer segments effectively.

Database Design

Country Table:

COUNTRY_ID: Numeric identifier for a country. (Primary Key)

COUNTRY: Name of the country.

Customer Table:

CUSTOMER_ID: Numeric identifier for a customer. (Primary Key)

FIRST_NAME: First name of the customer.

LAST_NAME: Last name of the customer.

GENDER_ID: Numeric identifier representing the gender of the customer. (Foreign Key)

AGE: Age of the customer.

ANNUAL_INCOME_K\$: Annual income of the customer in thousands.

SPENDING_SCORE: Customer's spending score.

COUNTRY_ID: Numeric identifier representing the country of the customer. (Foreign Key)

Customer Staging Table:

CUSTOMERID: Numeric identifier for a customer.

GENDER: Gender of the customer.

AGE: Age of the customer.

ANNUAL_INCOME_THOUSANDS: Annual income of the customer in thousands.

SPENDING_SCORE: Customer's spending score.

Gender Table:

GENDER_ID: Numeric identifier for a gender. (Primary Key)

GENDER: Gender description (e.g., Male, Female)

Invoice Table:

INVOICE_ID: Numeric identifier for an invoice. (Primary Key)

INVOICE_DATE: Date of the invoice.

INVOICE_NO: Invoice number.

CUSTOMER_ID: Numeric identifier representing the customer associated with the invoice. (Foreign Key)

TOTAL_AMOUNT: Total amount of the invoice.

Invoice Line Item Table:

INVOICE_LINE_ID: Numeric identifier for an invoice line item. (Primary Key)

INVOICE_ID: Numeric identifier representing the invoice associated with the line item. (Foreign Key)

PRODUCT_ID: Numeric identifier representing the product in the line item. (Foreign Key)

QUANTITY: Quantity of the product in the line item.

LINE_TOTAL: Total cost of the line item.

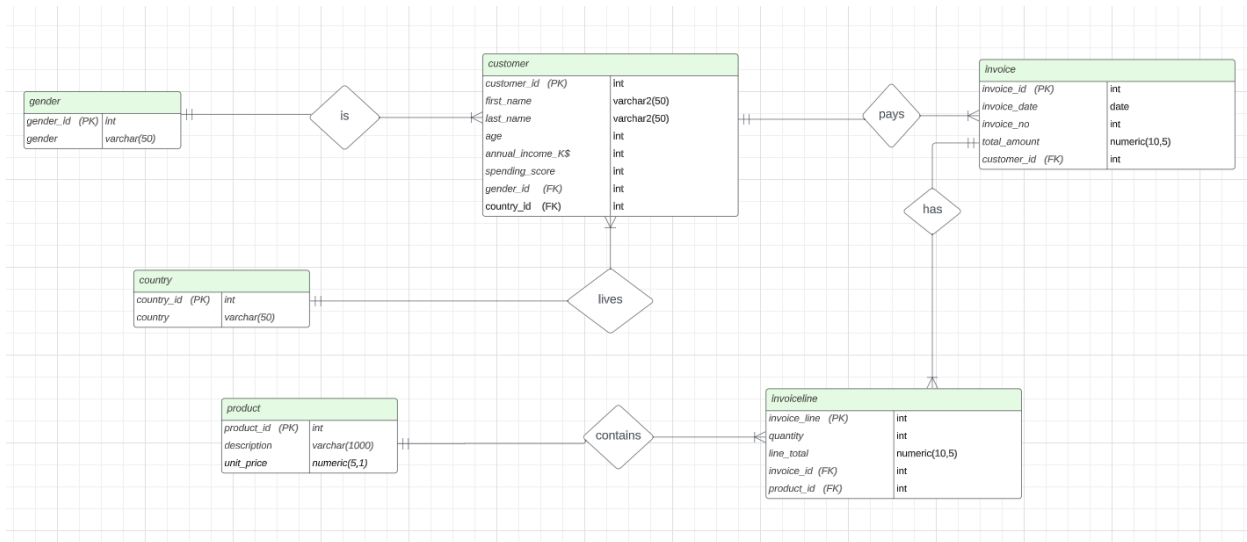
Product Table:

PRODUCT_ID: Numeric identifier for a product. (Primary Key)

DESCRIPTION: Description of the product.

UNIT_PRICE: Price of the product per unit.

Entity Relationship Diagram



Establishing the relationship

Creating a strong connection between our two datasets is vital for building an effective e-commerce database. We've done this by using a unique identifier, known as "CustomerID," which helps us link each transaction in the "Online Retail" dataset to the specific customer who made the purchase. This link allows us to understand the buying habits of individual customers, track what they've bought before, and offer them personalized services and recommendations.

By having this relationship, our database can do things like keeping a record of what each customer has ordered, managing billing and invoices, and providing better customer service. In simple terms, this connection between the "Online Retail" and "Customer" datasets is crucial for making our e-commerce platform work smoothly and provide customers with a more personalized and satisfying shopping experience.

Data Dictionaries

By establishing the relation between the datasets OnlineRetail and Customer, the following tables were created:

Country Table:

COUNTRY_ID: Numeric identifier for a country. (Primary Key)

COUNTRY: Name of the country.

Customer Table:

CUSTOMER_ID: Numeric identifier for a customer. (Primary Key)

FIRST_NAME: First name of the customer.

LAST_NAME: Last name of the customer.

GENDER_ID: Numeric identifier representing the gender of the customer. (Foreign Key)

AGE: Age of the customer.

ANNUAL_INCOME_K\$: Annual income of the customer in thousands.

SPENDING_SCORE: Customer's spending score.

COUNTRY_ID: Numeric identifier representing the country of the customer. (Foreign Key)

Customer Staging Table:

CUSTOMERID: Numeric identifier for a customer.

GENDER: Gender of the customer.

AGE: Age of the customer.

ANNUAL_INCOME_THOUSANDS: Annual income of the customer in thousands.

SPENDING_SCORE: Customer's spending score.

Gender Table:

GENDER_ID: Numeric identifier for a gender. (Primary Key)

GENDER: Gender description (e.g., Male, Female).

Invoice Table:

INVOICE_ID: Numeric identifier for an invoice. (Primary Key)

INVOICE_DATE: Date of the invoice.

INVOICE_NO: Invoice number.

CUSTOMER_ID: Numeric identifier representing the customer associated with the invoice. (Foreign Key)

TOTAL_AMOUNT: Total amount of the invoice.

Invoice Line Item Table:

INVOICE_LINE_ID: Numeric identifier for an invoice line item. (Primary Key)

INVOICE_ID: Numeric identifier representing the invoice associated with the line item. (Foreign Key)

PRODUCT_ID: Numeric identifier representing the product in the line item. (Foreign Key)

QUANTITY: Quantity of the product in the line item.

LINE_TOTAL: Total cost of the line item.

Product Table:

PRODUCT_ID: Numeric identifier for a product. (Primary Key)

DESCRIPTION: Description of the product.

UNIT_PRICE: Price of the product per unit.

Tables Creation and Data Loading

Create Queries

Country Table

```
create table country (  
country_id NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,  
country varchar(50)  
);
```

Gender Table

```
create table gender (  
gender_id NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,  
gender varchar(50)  
);
```

Product Table

```
create table product (  
product_id NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,  
stock_code int,  
description varchar(1000),  
unit_price numeric(10,5)  
);
```

Customer Table

```
create table customer (  
customer_id NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,  
first_name VARCHAR2(50),  
last_name VARCHAR2(50),  
gender_id int,  
age int,  
annual_income_K$ int,  
spending_score int,  
country_id int,  
CONSTRAINT fk_gender_id  
FOREIGN KEY (gender_id)  
REFERENCES gender (gender_id),  
CONSTRAINT fk_country_id  
FOREIGN KEY (country_id)  
REFERENCES country (country_id) );
```

Invoice Table

```
create table invoice (  
  invoice_id NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,  
  invoice_date date,  
  invoice_no int,  
  customer_id int ,  
  total_amount numeric(10,5),  
  CONSTRAINT fk_customer_id  
    FOREIGN KEY (customer_id)  
    REFERENCES customer (customer_id)  
  
);
```

Invoice Line Item Table

```
create table invoice_line_item (  
  invoice_line_id NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,  
  invoice_id int,  
  product_id int ,  
  quantity int,  
  line_total numeric(15,5),  
  CONSTRAINT fk_invoice_id  
    FOREIGN KEY (invoice_id)  
    REFERENCES invoice (invoice_id),  
  CONSTRAINT fk_product_id  
    FOREIGN KEY (product_id)  
    REFERENCES product (product_id)  
  
);
```

C-I Table

```
create table c_i(  
  customerid int,  
  invoiveno varchar(50)  
  
);
```

Inserting Data into tables

Inserting into country

```
INSERT INTO country (country)  
select distinct country from online_retail ;
```


Inserting into gender

```
INSERT INTO gender (gender)
select distinct gender from customer_staging where gender is not null;
```

Inserting into product

```
INSERT INTO product (description,unit_price)
select distinct description,unitprice from online_retail;
```

```
BEGIN
  DBMS_RANDOM.seed(42);

END;
/
```

Inserting into customer

```
INSERT INTO customer (first_name, last_name, gender_id, age, ANNUAL_INCOME_K$, spending_score, country_id)
SELECT
  DBMS_RANDOM.STRING('A', 10) AS first_name,
  DBMS_RANDOM.STRING('A', 10) AS last_name,
  g.gender_id,
  cs.age,
  cs.ANNUAL_INCOME_THOUSANDS,
  cs.spending_score,
  c.country_id -- Assuming the table 'country' has a record with this ID.
FROM customer_staging cs
INNER JOIN gender g ON g.gender = cs.gender
CROSS JOIN (SELECT country_id FROM country
            ORDER BY DBMS_RANDOM.VALUE(1,32)) c;
```

Inserting into c_i

```
insert into c_i(customerid,invoiveno)
select FLOOR(DBMS_RANDOM.VALUE(1, 200)) as customer_id, invoiceno from (select distinct invoiceno from
online_retail );
```

```
UPDATE online_retail
SET customerid = (select customerid from c_i where c_i.invoiveno = online_retail.invoiceno );
```

Inserting into invoice

```
INSERT INTO invoice (invoice_date,invoice_no ,customer_id, total_amount)
SELECT
  invoicedate,
  invoiceno,
```

```
customerid,  
1.00--floor(sum(cost_of_one_item)) as total_amount  
FROM  
(select invoicedate,invoiceno,customerid,quantity * unitprice as cost_of_one_item from online_retail)  
group by invoicedate, invoiceno, customerid;
```

Inserting into invoice_line_item

```
INSERT INTO invoice_line_item (invoice_id,product_id ,quantity, line_total)  
select i.invoice_id,p.product_id,quantity,quantity * unitprice as line_total from online_retail o  
inner join invoice i on i.invoice_no = o.invoiceno  
inner join product p on o.description = p.description
```

Performance tuning

```
CREATE INDEX invoiceno_index  
ON online_retail (invoiceno);
```

```
CREATE INDEX description_index  
ON online_retail (description);
```

Experiment: Indexing Strategy Impact on Query Performance

Purpose of the Experiment:

The purpose of this experiment was to assess the effect of indexing on the performance of a complex SQL query that involves multiple joins across several tables. The hypothesis was that indexing the columns used in the JOIN conditions would reduce the query execution time by allowing the database engine to locate and retrieve related data more efficiently.

Steps Followed to Run the Experiment:

A complex query was identified, which joins the invoice, product, and online_retail tables and computes a line_total using columns from these tables.

The query execution time was measured without any additional indexes, serving as the baseline performance metric.

Indexes were then created on the joining columns across the tables involved in the query.

The same query was executed again, and the execution time was recorded post-indexing.

Key Results:

Before indexing, the query fetched 50 rows in 0.259 seconds.

After indexing, the query fetched 50 rows in 0.117 seconds.

Discussion of the Results:

The results clearly indicate a significant improvement in query performance due to indexing. The query execution time was reduced by more than 50%, demonstrating the effectiveness of indexes in optimizing SQL query performance. The performance gain can be attributed to the reduced need for full table scans, as the database engine could quickly locate the necessary rows using the indexes. This is particularly beneficial for the JOIN operations, as the database engine can use the index to match rows between tables more efficiently than scanning all rows. The improvement in I/O burden due to indexing likely contributed to faster data retrieval, as observed in the reduced execution time.

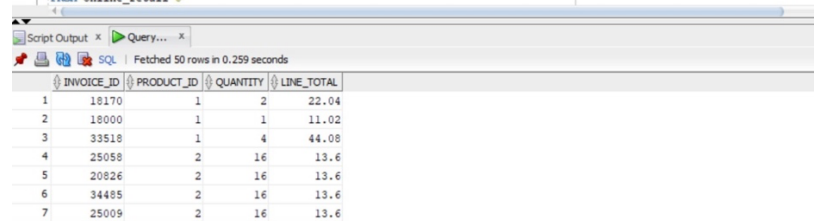
These results support the best practice of creating indexes on columns that are frequently used in JOIN conditions, especially for complex queries involving multiple tables. It should be noted that while indexes can dramatically

improve read operations, they may introduce overhead for write operations due to the additional updates needed for the index structures. Therefore, the decision to index should be balanced based on the read and write workload of the database.

Before indexing

```
select i.invoice_id,p.product_id,quantity,quantity * unitprice as line_total from online_retail o
inner join invoice i on i.invoice_no = o.invoiceno and i.invoice_date = o.invoicedate
inner join product p on o.stockcode = p.stock_code and o.unitprice = p.unit_price and o.description = p.description ;

SELECT count(*)
FROM online_retail o
```

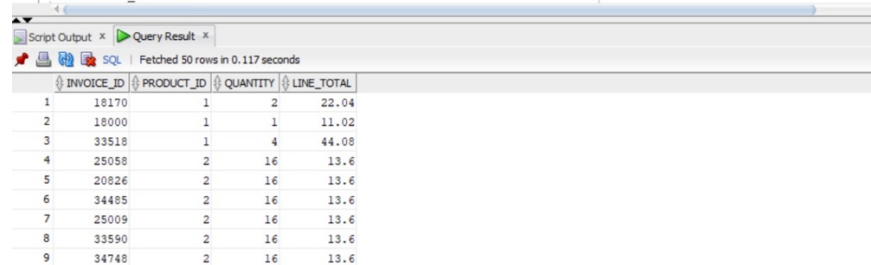


INVOICE_ID	PRODUCT_ID	QUANTITY	LINE_TOTAL
1	18170	1	22.04
2	18000	1	11.02
3	33518	1	44.08
4	25058	2	13.6
5	20826	2	13.6
6	34485	2	13.6
7	25009	2	13.6

After Indexing

```
select i.invoice_id,p.product_id,quantity,quantity * unitprice as line_total from online_retail o
inner join invoice i on i.invoice_no = o.invoiceno and i.invoice_date = o.invoicedate
inner join product p on o.stockcode = p.stock_code and o.unitprice = p.unit_price and o.description = p.description ;

SELECT count(*)
FROM online_retail o
```



INVOICE_ID	PRODUCT_ID	QUANTITY	LINE_TOTAL
1	18170	1	22.04
2	18000	1	11.02
3	33518	1	44.08
4	25058	2	13.6
5	20826	2	13.6
6	34485	2	13.6
7	25009	2	13.6
8	33590	2	13.6
9	34748	2	13.6

2. Experiment: Impact of Optimizer Changes on Query Performance

Purpose of the Experiment:

The objective of this experiment was to determine the effect of optimizer changes on the execution time of a SQL query involving joins across several tables. The expectation was that modifying the optimizer settings could lead to a more efficient execution plan, thus reducing the query execution time.

Steps Followed to Run the Experiment:

A specific query was identified that joins the invoice, product, and online_retail tables.

The query's execution time was recorded to establish a baseline for performance measurement.

The optimizer setting was then changed to a different mode (details of the specific change were not provided but might involve changing from rule-based to cost-based optimization, or adjusting optimizer parameters).

The same query was executed again after the optimizer change, and the new execution time was recorded.

Key Results:

Before the optimizer change, the query fetched 50 rows in 1.838 seconds.

After the optimizer change, the query fetched 50 rows in 0.892 seconds.

Discussion of the Results:

The results exhibit a substantial improvement in query performance following the optimizer changes, with the execution time more than halving from 1.838 seconds to 0.892 seconds. This suggests that the optimizer was able to generate a more efficient execution plan for the query after the adjustments.

The significant reduction in time could be due to several factors influenced by the optimizer's mode, such as:

- Improved join order that minimized the number of rows processed.

- More effective use of indexes if available.

- Better estimation of statistics leading to more accurate cardinality and cost estimates.

The exact reasons for the performance improvement would typically be verified by examining the execution plans before and after the optimizer change. This would reveal changes in access paths (such as index scans vs. full table scans), join methods (nested loops, hash joins, etc.), and join orders.

These findings underscore the importance of proper database tuning and the potential impact of optimizer settings on query performance. However, it's also crucial to consider the broader implications of such changes, as different queries may respond differently to optimizer adjustments. What benefits one query might negatively affect another, so comprehensive testing and analysis are essential.

Before changing the optimizer

```
select i.invoice_id,p.product_id,quantity,quantity * unitprice as line_total from online_retail o
inner join invoice i on i.invoice_no = o.invoiceno and i.invoice_date = o.invoicedate
inner join product p on o.stockcode = p.stock_code and o.unitprice = p.unit_price and o.description = p.description ;
```

SELECT count(*)

Script Output x Query Result x

SQL | Fetched 50 rows in 1.838 seconds

	INVOICE_ID	PRODUCT_ID	QUANTITY	LINE_TOTAL
1	18170	1	2	22.04
2	18000	1	1	11.02
3	33518	1	4	44.08
4	25009	2	16	13.6
5	25058	2	16	13.6
6	20826	2	16	13.6
7	33590	2	16	13.6
8	33812	2	5	4.25
9	34485	2	16	13.6

After changing the optimizer

```
--before optimizer

select i.invoice_id,p.product_id,quantity,quantity * unitprice as line_total from online_retail o
inner join invoice i on i.invoice_no = o.invoiceno and i.invoice_date = o.invoicedate
inner join product p on o.stockcode = p.stock_code and o.unitprice = p.unit_price and o.description = p.description
```

	INVOICE_ID	PRODUCT_ID	QUANTITY	LINE_TOTAL
1	18170	1	2	22.04
2	18000	1	1	11.02
3	33518	1	4	44.08
4	25058	2	16	13.6
5	20826	2	16	13.6
6	34485	2	16	13.6
7	25009	2	16	13.6
8	33590	2	16	13.6
9	34748	2	16	13.6

Query Writing

Identifying the most popular product based on the quantity sold. :

```
SELECT product.description, SUM(invoice_line_item.quantity) AS total_quantity_sold
FROM product
JOIN invoice_line_item ON product.product_id = invoice_line_item.product_id
GROUP BY product.description
ORDER BY total_quantity_sold DESC;
```

```
SELECT product.description, SUM(invoice_line_item.quantity) AS total_quantity_sold
FROM product
JOIN invoice_line_item ON product.product_id = invoice_line_item.product_id
GROUP BY product.description
ORDER BY total_quantity_sold DESC;
```

	DESCRIPTION	TOTAL_QUANTITY_SOLD
1	MEDIUM CERAMIC TOP STORAGE JAR	74639

Determine the average spending score of customers by gender.:

```
SELECT gender.gender, AVG(customer.spending_score) AS average_spending_score
FROM gender
JOIN customer ON gender.gender_id = customer.gender_id
GROUP BY gender.gender;
```

[illegible]

```
SELECT first_name, last_name, annual_income_K$, spending_score
FROM customer
WHERE spending_score > 80
ORDER BY annual_income_K$ DESC;
```

Calculate the total number of invoices per month.:

	FIRST_NAME	LAST_NAME	ANNUAL_INCOME_K\$	SPENDING_SCORE
1	bmOtJGrYJj	bDTAxkpDIU	137	83
2	hWEcevgWpO	pPpjvXmVDN	113	91
3	DPPaiTnPzz	UXMLNjxpRz	103	85
4	EmugjJUMoQ	uYPbAGhgAj	99	97
5	MjMwyHaVtS	fXvlRajKEj	98	88
6	OvMDDXsCie	pEIAhuHKLi	97	86
7	zffAAAdHhpc	PHmDGFpLuU	93	90
8	ydxufhxtYF	nxWeJzPcHj	88	86
9	WyIdjErNoo	uTPDKbvLfy	87	92
10	gstnlxVAiI	YexGTRmROU	86	95
11	HBiafbmJvb	SinICnrDqW	81	93
12	XJLNFTbwvZ	stNhjYPWoD	79	83
13	kmOQSaLoeT	elzTUoidSb	78	89
14	sSzQBicfYm	XUcgzUISBb	78	88
15	YfMgoFpRCw	dSsRlswOwH	78	90

```
SELECT EXTRACT(YEAR FROM invoice_date), EXTRACT(MONTH FROM invoice_date), COUNT(*) AS total_invoices
FROM invoice
GROUP BY EXTRACT(YEAR FROM invoice_date), EXTRACT(MONTH FROM invoice_date)
```

ORDER BY EXTRACT(YEAR FROM invoice_date), EXTRACT(MONTH FROM invoice_date);

```

SELECT EXTRACT(YEAR FROM invoice_date), EXTRACT(MONTH FROM invoice_date), COUNT(*) AS total_invoices
FROM invoice
GROUP BY EXTRACT(YEAR FROM invoice_date), EXTRACT(MONTH FROM invoice_date)
ORDER BY EXTRACT(YEAR FROM invoice_date), EXTRACT(MONTH FROM invoice_date);

```

Query Result x

SQL | All Rows Fetched: 13 in 0.035 seconds

	EXTRACT(YEARFROMINVOICE_DATE)	EXTRACT(MONTHFROMINVOICE_DATE)	TOTAL_INVOICES
1	2010	12	1341
2	2011	1	1006
3	2011	2	983
4	2011	3	1330
5	2011	4	1143
6	2011	5	1491
7	2011	6	1357
8	2011	7	1300
9	2011	8	1226
10	2011	9	1668
11	2011	10	1847
12	2011	11	2478
13	2011	12	720

List products that have never been sold. :

```

SELECT product_id, description
FROM product
WHERE product_id NOT IN (SELECT DISTINCT product_id FROM invoice_line_item);

```

```

--List products that have never been sold. :

SELECT product_id, description
FROM product
WHERE product_id NOT IN (SELECT DISTINCT product_id FROM invoice_line_item);

```

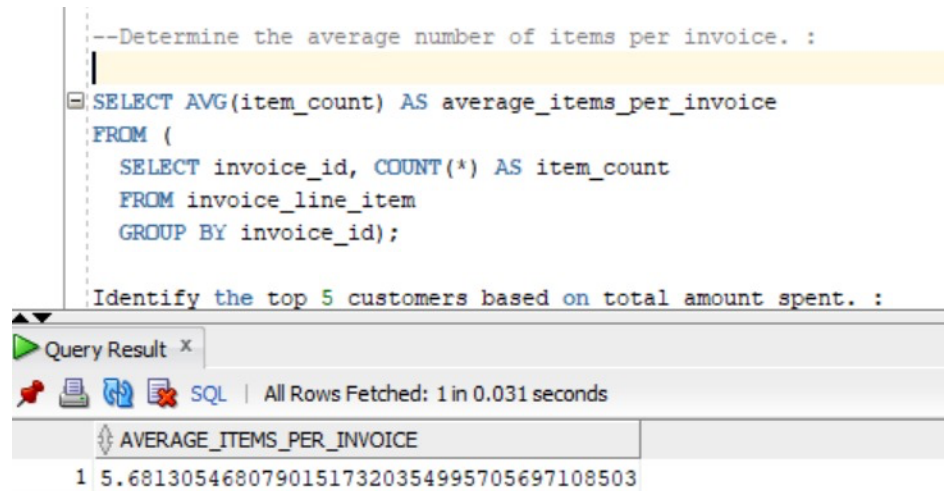
Query Result x

SQL | Fetched 50 rows in 0.07 seconds

	PRODUCT_ID	DESCRIPTION
1	46	POSTAGE
2	194	DOTCOM POSTAGE
3	270	(null)
4	301	(null)
5	376	POSTAGE
6	392	POSTAGE
7	393	DOTCOM POSTAGE
8	442	DOTCOM POSTAGE
9	461	(null)
10	485	Manual
11	504	DOTCOM POSTAGE
12	567	(null)
13	578	Discount
14	579	Manual

Determine the average number of items per invoice. :

```
SELECT AVG(item_count) AS average_items_per_invoice
FROM (
  SELECT invoice_id, COUNT(*) AS item_count
  FROM invoice_line_item
  GROUP BY invoice_id);
```



The screenshot shows a SQL query editor with a yellow highlight on the first line of the query. Below the editor, a 'Query Result' window displays the results of the query. The window shows a single row with the column name 'AVERAGE_ITEMS_PER_INVOICE' and the value '1 5.68130546807901517320354995705697108503'.

```
--Determine the average number of items per invoice. :
SELECT AVG(item_count) AS average_items_per_invoice
FROM (
  SELECT invoice_id, COUNT(*) AS item_count
  FROM invoice_line_item
  GROUP BY invoice_id);
```

Identify the top 5 customers based on total amount spent. :

AVERAGE_ITEMS_PER_INVOICE
1 5.68130546807901517320354995705697108503

Identifying the top 5 customers based on total amount spent. :

```
SELECT customer.customer_id, first_name, last_name, SUM(invoice.total_amount) AS total_spent
FROM customer
JOIN invoice ON customer.customer_id = invoice.customer_id
GROUP BY customer.customer_id, first_name, last_name
ORDER BY total_spent DESC;
```



```
--Identify the top 5 customers based on total amount spent. :
```

```
SELECT
    customer.customer_id,
    first_name,
    last_name,
    SUM(invoice.total_amount) AS total_spent
FROM
    customer
```

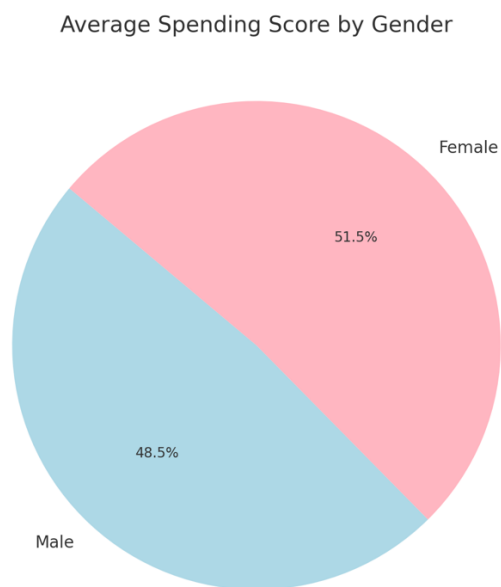
Query Result x

SQL | Fetched 50 rows in 0.229 seconds

	CUSTOMER_ID	FIRST_NAME	LAST_NAME	TOTAL_SPENT
1	53	lcpyRiaQyN	tWLhGTjnNs	88822.68
2	137	YALLvxcbLv	cSOyhIdEPK	32673.16
3	38	DPVbpTF1QK	xNTJtCWAqY	20626.64
4	35	uiEHcOPCEn	YeRcyCDFRc	17747.4
5	3	LSkmpNqBeA	PammGtFCdG	17358.78

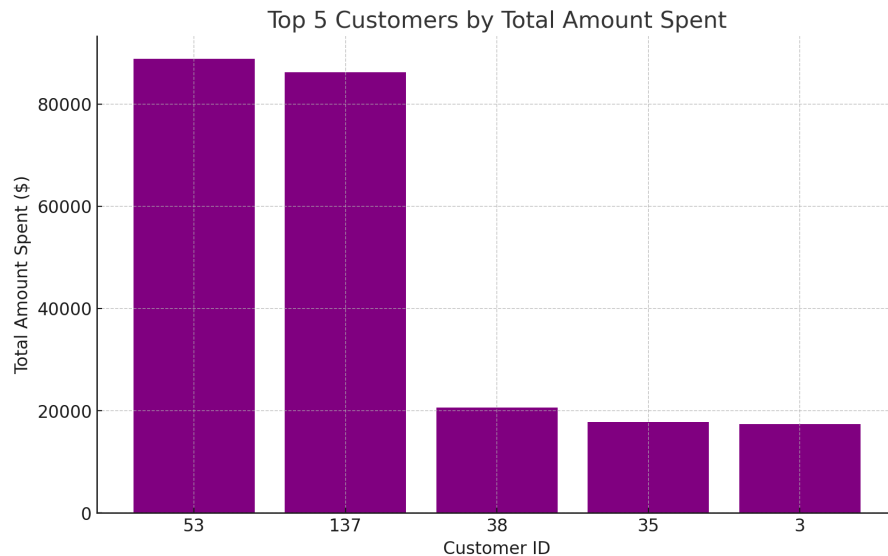
Tableau Visualization

Visualizing the average spending score of customers by gender.:



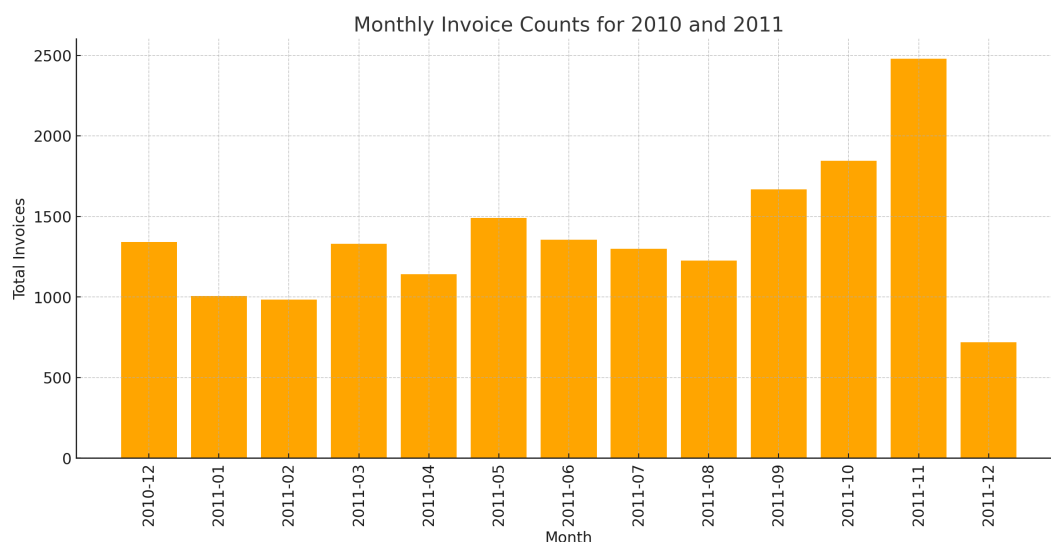
The pie chart depicts the average spending score by gender, showing a close distribution with females having a slightly higher average spending score at 51.5% compared to males at 48.5%. This visual representation suggests a marginal difference in spending behavior between the genders.

Identifying the top 5 customers based on total amount spent



The bar graph illustrates the spending of the top five customers, with customer '53' leading significantly in total expenditures, followed by customer '137'. Customers '38', '35', and '3' show relatively similar spending patterns, with a substantial decrease from the top two spenders.

Visualizing the total number of invoices per month



The bar graph represents the monthly count of invoices for the years 2010 and 2011, showing a fluctuating trend in invoice volume across the months. Notably, November 2011 experienced the highest number of invoices, while December 2011 saw a significant drop, possibly due to seasonal business cycles.

