Online Retail Transactional Database

A detailed description of the DBMS:

Oracle is a powerful, object-relational database management system (DBMS) produced and marketed by Oracle Corporation. It is one of the world's most popular databases for running enterprise-level applications and has been a leader in the database sector for several decades.

As a DBMS, Oracle is known for its robust feature set, which supports large-scale transactions, parallel processing, and a wide array of business operations. The system uses SQL as its primary data manipulation language, and PL/SQL, Oracle's procedural extension to SQL, allowing for stored procedures and functions, triggers, and advanced data manipulation.

Advantages of Oracle over other DBMS:

Oracle Database is a widely used relational database management system (RDBMS) that has some advantages over other database systems like MySQL, PostgreSQL, SQLite, IBM Db2 on Cloud, Microsoft Azure SQL, MariaDB, Amazon Redshift, Redis, Google BigQuery, Google AlloyDB for PostgreSQL, Amazon Aurora, and SQLite. Here are some advantages of Oracle Database:

1. Scalability: Oracle Database is known for its ability to handle large-scale and enterprise-level workloads. It provides options for partitioning, clustering, and data sharding, making it suitable for high-demand applications.

2. High Availability: Oracle offers various features and technologies for achieving high availability, including Oracle Real Application Clusters (RAC), Data Guard, and automatic failover capabilities, ensuring minimal downtime.

3. Security: Oracle Database has a robust security model with features like encryption, auditing, and fine-grained access control. It complies with various security standards and regulations.

4. Advanced Analytics: Oracle Database provides support for advanced analytics and data processing through features like in-database analytics, machine learning, and spatial data capabilities.

5. Integration: Oracle Database integrates well with other Oracle products and technologies, making it suitable for organizations with a comprehensive Oracle ecosystem.

6. Enterprise Support: Oracle offers extensive enterprise-level support and services, making it a reliable choice for organizations that require top-tier support and assistance.

7. Data Warehousing: Oracle Database provides features for building and managing data warehouses, making it a strong contender for business intelligence and analytics use cases.

8. Advanced Query Optimization: Oracle Database has a sophisticated query optimizer that can handle complex queries efficiently, which is crucial for performance-critical applications.

9. Compatibility: Oracle Database supports standard SQL and offers compatibility features for migrating from other database systems.

10. Backup and Recovery: Oracle Database offers robust backup and recovery solutions, ensuring data protection and disaster recovery capabilities.

OLTP in Oracle:

Oracle is widely used for OLTP systems, which are characterized by a large number of short online transactions (INSERT, UPDATE, DELETE). The main emphasis for OLTP systems is put on very fast query processing, maintaining data integrity in multi-access environments, and an effectiveness measured by the number of transactions per second.

In OLTP databases, speed of transaction processing is critical, and Oracle achieves this through a combination of in-memory data storage, optimized network protocols, and efficient transaction management. Oracle's architecture separates transaction processing from query processing, which allows for concurrent processing without locking users out of the system.

Oracle's ACID-compliant transaction model ensures that when data is manipulated, all transactions are processed reliably, and concurrent transaction processing does not lead to data inconsistency. Oracle’s Multi-Version Concurrency Control (MVCC) allows the system to maintain consistent snapshots for read operations, even as data is being changed, ensuring the stability of OLTP operations.

To sum up, Oracle’s combination of performance, scalability, reliability, security, and robust support for transaction processing makes it a leading choice for enterprises that require an efficient and reliable OLTP database system.

A detailed description of the KDD Nuggets referenced data

We choose 2 datasets from KKD nuggets for the project

In the process of building an e-commerce transactional database, we are working with two key datasets: the "Online Retail" dataset and the "Customer" dataset. These datasets are integral to the establishment of a comprehensive and efficient e-commerce database system, and we are focusing on establishing a robust relationship between them.

Dataset 1: Online Retail

Source Link: https://archive.ics.uci.edu/dataset/352/online+retail

Donation Date: 11/5/2015

Data Type: Transnational

Data Description:

This dataset contains information on all transactions occurring between 01/12/2010 and 09/12/2011 for a UK-based and registered non-store online retail company.

The company primarily sells unique all-occasion gifts, and many of its customers are wholesalers.

Feature	Type of Variable	Values
InvoiceNo	Nominal	6-digit integer
StockCode	Nominal	5-digit integer
Description	Nominal	Strings (names)
Quantity	Numeric	Integer values
InvoiceDate	Numeric	Date and time
UnitPrice	Numeric	float
CustomerID	Numeric	5-digit integer
Country	Nominal	Strings(name)

Dataset Characteristics:

Multivariate: The dataset contains multiple variables.

Sequential: The data is organized sequentially, likely based on transaction timestamps.

Time-Series: The dataset involves time-series data, as it captures transactions over a period.

Subject Area: Business

Dataset Size:

Number of Instances: 541,909

Number of Features: 6

Additional Information: The dataset does not contain missing values.

Variables:

InvoiceNo: This is a categorical variable serving as an ID for each transaction. It is a 6-digit integral number uniquely assigned to each transaction. Transactions starting with the letter 'c' indicate cancellations.

StockCode: Another categorical variable representing a product (item) code. It is a 5-digit integral number uniquely assigned to each distinct product.

Description: This is a categorical variable containing the product name.

Quantity: An integer feature representing the quantities of each product (item) per transaction.

InvoiceDate: A date feature indicating the day and time when each transaction was generated.

UnitPrice: A continuous feature representing the product's price per unit in sterling.

CustomerID: A categorical variable, a 5-digit integral number uniquely assigned to each customer.

Country: Another categorical variable representing the name of the country where each customer resides.

Dataset 2:

SourceLink: https://www.kaggle.com/datasets/shrutimechlearn/customer-data

Data Type: Demographic and Behavioral

Data Description:

This dataset includes information on customers' demographics and behavior, specifically focusing on their purchasing patterns. It is used for market segmentation purposes to understand customer traits and to tailor marketing strategies accordingly. The data encapsulates various aspects of customer profiles, including gender, age, income, and a constructed "Spending Score" likely reflecting the customers' purchasing behavior or frequency.

Dataset Characteristics:

Univariate & Multivariate: The dataset holds multiple variables for each customer, but each variable can also be analyzed individually.

Cross-Sectional: The data is static and represents a snapshot of customers at a given time.

Associative: The dataset allows for the analysis of relationships between different variables such as age and spending score.

Subject Area: Marketing, Retail, Consumer Behavior

Dataset Size:

Number of Instances: 200

Number of Features: 5 (excluding the CustomerID)

Additional Information: The dataset does not contain missing values

Variables:

Feature	Type of Variable	Values
CustomerID	Numeric	integer
Age	Numeric	integer
Genre	Nominal	Strings (gender)
Annual_Income_(k$)	Numeric	float
Spending_Score	Numeric	int

CustomerID: A categorical variable serving as a unique identifier for each customer.

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

Age: A numerical variable representing the age of the customer.

Annual_Income_(k$): A numerical variable indicating the customer's annual income in thousands of dollars. This is a key variable for understanding purchasing power.

Spending_Score: A numerical score assigned to the customer based on their purchasing behavior and patterns. This is a constructed variable likely based on proprietary algorithms or business rules.

Building the Relationship:

To create a coherent and transactional e-commerce database, it's crucial to establish a strong relationship between these two datasets. This connection is typically accomplished by using a unique identifier, often referred to as a "CustomerID" or "UserID."

The shared identifier links each transaction recorded in the "Online Retail" dataset to the specific customer who made the purchase. This connection allows the e-commerce system to attribute each transaction to a customer.

With this relationship, the database can provide insights into the buying habits of individual customers, track their purchase history, and facilitate personalized services, such as recommendations or targeted marketing.

Furthermore, it allows the system to manage order histories for customers, manage billing and invoicing, and support customer service functions more effectively.

In summary, the synergy between the "Online Retail" dataset and the "Customer" dataset is pivotal in constructing a comprehensive e-commerce transactional database. It facilitates the organization and management of transactions while enabling the system to understand and serve its customers better through personalized services and insights into customer behavior and preferences. This integrated approach is essential for a successful and customer-centric e-commerce platform.

Detailed Product Description: Transactional Design Rationale

Our product is an e-commerce transactional database designed to handle and record all transactions that occur through an e-commerce platform. This database is structured to manage a vast amount of data that reflects the dynamic nature of online retail activities. It's created to serve as the backbone of an e-commerce business, ensuring the integrity, availability, and consistency of transaction data in real-time.

Why the Design Makes it Transactional:

ACID Properties:

The database is designed to adhere to the ACID (Atomicity, Consistency, Isolation, Durability) properties, which are essential for transactional systems. This means that each transaction is treated as a single unit of work, changes are made consistently, transactions do not interfere with each other, and once a transaction is committed, it will remain so, even in the event of a system failure.

Real-Time Processing:

Transactions are processed in real-time, ensuring that data such as inventory levels, customer orders, and financial records are always up to date. This immediate processing is crucial for an e-commerce platform where delays can result in poor customer experiences or financial discrepancies.

Concurrent Access:

The database is designed for high concurrency, allowing multiple transactions to occur simultaneously without conflict. This is vital for e-commerce websites that deal with multiple customers and transactions at the same time.

Data Integrity and Security:

Ensuring data integrity involves maintaining the accuracy and consistency of data over its entire lifecycle. The database is equipped with constraints, triggers, and authorization controls to safeguard data integrity and enforce business rules. Security measures such as encryption and access control are also in place to protect sensitive transaction data against unauthorized access or breaches.

Scalability:

As the e-commerce business grows, the database can scale to handle increasing transaction volumes without degradation in performance. This means implementing scalable architectures like sharding or partitioning to distribute the load and optimize performance.

Logging and Audit Trails:

The database keeps logs and audit trails of all transactions, enabling the tracking of changes and supporting the auditability of the system. This is necessary for compliance with financial regulations and for resolving any disputes that may arise.

Robust Backup and Recovery:

To prevent transaction loss, the database includes robust backup and recovery mechanisms. It ensures that the system can quickly recover from hardware failures, power outages, or other unexpected issues without data loss.

Support for Complex Transactions:

The database supports complex transactions, which may include multiple steps or stages. For example, an e-commerce transaction could involve inventory checks, payment processing, and order confirmation, all of which need to be completed successfully for the transaction to be considered complete.

In conclusion, the e-commerce transactional database is engineered specifically to manage, store, and secure transactions for online retail businesses. Its transactional nature is characterized by the ability to handle many quick, simultaneous transactions while maintaining data integrity, consistency, and reliability, which are all critical for the success of an e-commerce platform.

Product Data Structures: Description

Tables:

Tables are database objects that store data in rows and columns and are the same elements we used and dealt with in mssql or sql server in class.

Create Tables syntax:

```
CREATE TABLE table_name
(
   column1 datatype,
   column2 datatype,
   ...
   CONSTRAINT constraint_name PRIMARY KEY (column_name),
   CONSTRAINT constraint_name UNIQUE (column_name),
   CONSTRAINT constraint_name FOREIGN KEY (column_name) REFERENCES
reference_table (reference_column));
```

- ***table_name:*** *The name of the table.*
- ***column1 datatype, column2 datatype, ...:*** *Define the table's columns and their data types.*
- ***CONSTRAINT constraint_name PRIMARY KEY (column_name):*** *Specifies a primary key constraint.*
- ***CONSTRAINT constraint_name UNIQUE (column_name):*** *Specifies a unique constraint.*
- ***CONSTRAINT constraint_name FOREIGN KEY (column_name) REFERENCES reference_table (reference_column):*** *Specifies a foreign key constraint.*

Sequences:

A sequence is a database object that generates unique values. It is often used to generate unique primary key values. We had to create these to store our latest primary key value for any table we create as an equivalent to 'Identity(1,1)' function in oracle to generate our primary keys.

Create sequence syntax:

```
CREATE SEQUENCE sequence_name
START WITH initial_value
INCREMENT BY increment_value
MAXVALUE max_value
MINVALUE min_value
NOCACHE/NOCYCLE;
```

- ***sequence_name:*** *The name of the sequence.*
- ***START WITH initial_value:*** *The initial value of the sequence.*
- ***INCREMENT BY increment_value:*** *The step size by which the sequence increases.*
- ***MAXVALUE max_value:*** *The maximum value the sequence can reach.*
- ***MINVALUE min_value:*** *The minimum value the sequence can reach.*
- ***NOCACHE:*** *This option disables caching of sequence values.*
- ***NOCYCLE:*** *This option prevents the sequence from cycling back to the minimum value after reaching the maximum value.*

Triggers:

A trigger is a set of actions that are automatically executed when a specific database event occurs, such as an INSERT, UPDATE, or DELETE operation. Triggers helped us increment our primary key values upon insertion into the table finally creating the effect of the 'identity(1,1)' we discussed in class and is used in mysql.

Create Triggers syntax:

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE/AFTER INSERT/UPDATE/DELETE ON table_name
FOR EACH ROW
BEGIN
   -- Trigger logic here
END;
```

- ***trigger_name:*** *The name of the trigger.*
- ***BEFORE/AFTER:*** *Specifies whether the trigger fires before or after the triggering event.*
- ***INSERT/UPDATE/DELETE:*** *Specifies the triggering event that activates the trigger.*
- ***table_name:*** *The name of the table to which the trigger is associated.*
- ***FOR EACH ROW:*** *Indicates that the trigger operates on each affected row.*
- ***BEGIN...END;****: Contains the trigger's PL/SQL logic.*

In these examples, replace `sequence_name`, `trigger_name`, `table_name`, `column_name`, `datatype`, and other placeholders with your specific names and values.

Indexes:

Indexes are database objects used to enhance query performance by providing fast access to data. We had problems dealing with approximately 500,000 values from our data

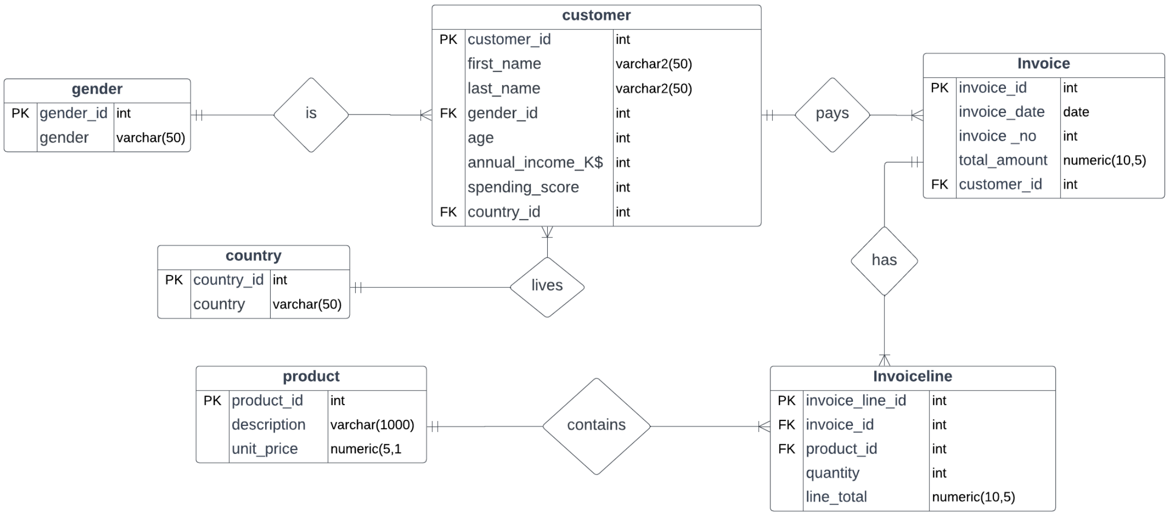
Create Index syntax:

```
CREATE INDEX index_name
ON table_name (column_name);
```

- ***index_name:*** *The name of the index.*
- ***table_name:*** *The name of the table on which the index is created.*
- ***column_name:*** *The column on which the index is built to optimize search and retrieval operations.*

Need for Index:

The dataset we are using has around 500,000 columns a few inner joins we performed took way to long to happen, so we had to create indexes on those columns to make the joins run faster.



Cardinality and Modality for ERD:

Customer to gender:

Assumption: Gender in the above dataset implies biological sex of the person

Cardinality: One customer has one gender; one gender can be associated with many customers. This is usually shown as a "one to many" relationship.

Modality: High modality on the side of the customer, indicating that the gender field for a customer is mandatory (each customer must have a gender). The gender entity, on the other hand, has low modality, indicating that not every gender needs to be assigned to a customer.

Customer to country:

Cardinality: One customer lives in one country; one country can have many customers. This is also a "one to many" relationship.

Modality: High modality on the customer side, each customer must be associated with a country. On the country side, it's low, indicating that there may be countries with no customers assigned.

Customer to invoice:

Cardinality: One customer can have many invoices; each invoice is paid by one customer. This is a "one to many" relationship.

Modality: High modality on the side of the invoice, because each invoice must be associated with a customer. On the customer side, the modality is low, meaning there can be customers without invoices.

Invoice to invoiceline:

Cardinality: One invoice can have many invoice lines, but each invoice line is associated with one invoice. This is a "one to many" relationship.

Modality: High on both sides, as an invoice should have at least one invoice line, and each invoice line must be associated with an invoice.

Product to invoiceline:

Cardinality: One product can be on many invoice lines; each invoice line refers to one product. This is a "one to many" relationship.

Modality: High modality on the invoiceline side (every invoiceline must be associated with a product), and low modality on the product side (not all products need to be on an invoice line).

Transactional CRUD Operations: A Detailed Description

Data definition Operations:

Create Statements:

Country Table:

- **Context**: The `country` table is designed to store information about the countries where customers, using an online retail platform, reside. This information is essential for understanding the geographical distribution of customers and their preferences.
- **Create Statement**:

```
CREATE TABLE country (
  country_id INT PRIMARY KEY,
  country VARCHAR(50)
);
CREATE SEQUENCE country_id_sequence
START WITH 1
INCREMENT BY 1
NOCACHE
NOCYCLE;

CREATE OR REPLACE TRIGGER country_id_trigger
BEFORE INSERT ON country
FOR EACH ROW
BEGIN
  SELECT country_id_sequence.NEXTVAL
  INTO :new.country_id
  FROM dual;
END;
/
```

- The `country` table consists of two columns: `country_id` and `country`.
- `country_id` serves as the primary key, ensuring each country has a unique identifier.
- `country` stores the name of the country where the customer resides.

Gender Table:

- **Context**: The `gender` table holds information about the gender of customers using the online retail platform. Understanding the gender distribution among customers can be valuable for marketing and product targeting.
- **Create Statement**:

```
CREATE TABLE gender (
  gender_id INT PRIMARY KEY,
  gender VARCHAR(50)
);
CREATE SEQUENCE gender_id_sequence
START WITH 1
INCREMENT BY 1
NOCACHE
NOCYCLE;

CREATE OR REPLACE TRIGGER gender_id_trigger
BEFORE INSERT ON gender
FOR EACH ROW
BEGIN
  SELECT gender_id_sequence.NEXTVAL
  INTO :new.gender_id
  FROM dual;
END;
/
```

- The `gender` table contains two columns: `gender_id` and `gender`.
- `gender_id` is the primary key, ensuring each gender category has a unique identifier.
- `gender` stores the gender descriptions (e.g., "Male" or "Female").

Product Table:

- **Context**: The `product` table stores information about the products available on the online retail platform. It includes details such as product names and unit prices.
- **Create Statement**:

```
CREATE TABLE product (
  product_id INT PRIMARY KEY,
  description VARCHAR(1000),
  unit_price NUMERIC(10, 5)
);
CREATE SEQUENCE product_id_sequence
START WITH 1
INCREMENT BY 1
NOCACHE
NOCYCLE;

CREATE OR REPLACE TRIGGER product_id_trigger
BEFORE INSERT ON product
FOR EACH ROW
BEGIN
  SELECT product_id_sequence.NEXTVAL
  INTO :new.product_id
  FROM dual;
END;
/
```

- The `product` table has three columns: `product_id`, `description`, and `unit_price`.

- `product_id` serves as the primary key, ensuring each product has a unique identifier.
- `description` stores the product name or description.
- `unit_price` represents the price per unit of the product in sterling.

Customer Table:

- **Context**: The `customer` table contains detailed information about customers using the online retail platform, including their demographic data, such as age, annual income, and spending score. It also links customers to their gender and country of residence.
- **Create Statement**:

```
CREATE TABLE customer (
  customer_id INT 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)
);
CREATE SEQUENCE customer_id_sequence
START WITH 1
INCREMENT BY 1
NOCACHE
NOCYCLE;

CREATE OR REPLACE TRIGGER customer_id_trigger
BEFORE INSERT ON customer
FOR EACH ROW
BEGIN
  SELECT customer_id_sequence.NEXTVAL
  INTO :new.customer_id
  FROM dual;
END;
/
```

- The `customer` table includes columns like `customer_id`, `first_name`, `last_name`, `gender_id`, `age`, `annual_income_K$`, `spending_score`, and `country_id`.
- `customer_id` serves as the primary key.
- `gender_id` and `country_id` are foreign keys, linking to the `gender` and `country` tables, respectively.
- Demographic data like age, annual income, and spending score provide insights into customer behavior and segmentation.

Invoice Table:

- **Context**: The `invoice` table records information related to customer transactions, including the invoice date, invoice number, customer, and total amount spent.
- **Create Statement**:

```
CREATE TABLE invoice (
  invoice_id INT 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)
);
CREATE SEQUENCE invoice_id_sequence
START WITH 1
INCREMENT BY 1
NOCACHE
NOCYCLE;

CREATE OR REPLACE TRIGGER invoice_id_trigger
BEFORE INSERT ON invoice
FOR EACH ROW
BEGIN
  SELECT invoice_id_sequence.NEXTVAL
  INTO :new.invoice_id
  FROM dual;
END;
/
```

- The `invoice` table contains columns for `invoice_id`, `invoice_date`, `invoice_no`, `customer_id`, and `total_amount`.
- `invoice_id` serves as the primary key.
- `customer_id` is a foreign key, connecting to the `customer` table.
- The table captures transaction details, including the date and total amount spent.

Invoice_line_item Table:

- **Context**: The `invoice_line_item` table stores detailed information about products within each invoice, including the product, quantity, and line total.
- **Create Statement**:

```
CREATE TABLE invoice_line_item (
  invoice_line_id INT 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)
);
CREATE SEQUENCE invoice_line_id_sequence
START WITH 1
INCREMENT BY 1
NOCACHE
NOCYCLE;

CREATE OR REPLACE TRIGGER invoice_line_id_trigger
BEFORE INSERT ON invoice_line_item
FOR EACH ROW
BEGIN
  SELECT invoice_line_id_sequence.NEXTVAL
  INTO :new.invoice_line_id
  FROM dual;
END;
/
```

- The `invoice_line_item` table comprises columns such as `invoice_line_id`, `invoice_id`, `product_id`, `quantity`, and `line_total`.
- `invoice_id` and `product_id` are foreign keys linking to the `invoice` and `product` tables, respectively.
- This table provides a detailed breakdown of items within each invoice.

c_i Table:

- **Context**: The `c_i` table is used for storing customer and invoice information, including `customerid` and `invoiveno`.
- This table captures data related to customer and invoice relationships.

These tables and triggers are part of a comprehensive database schema designed to manage data for a UK-based online retail platform. The data covers customer demographics, product details, transactions, and geographical information.

Index Statements:

- **Context**: Indexes have been created on the `online_retail` table to optimize queries involving `invoiceno` and `description` columns.
- **Index Statements**:

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

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

These tables, triggers, and indexes are part of a comprehensive database schema designed to manage data for a UK-based online retail platform. The data covers customer demographics, product details, transactions, and geographical information to support business analysis and decision-making.

Data Manipulation Operations: CRUD

Create :Insert Statements

Insert into `country` Table:

- **Context**: This insert statement populates the `country` table with distinct country names from a temporary table `online_retail` storing data from the ‘online retail’ uci dataset. The data represents the countries of customers using the UK-based non-store online retail platform.

Insert Statement:

```
INSERT INTO country (country)
SELECT DISTINCT country FROM online_retail;
```

Insert into `gender` Table:

- **Context**: This insert statement populates the `gender` table with distinct gender values from the temporary table `customer_staging` from the Kaggle ‘customer data’ dataset. The `genre` column represents gender information.

Insert Statement:

```
INSERT INTO gender (gender)
SELECT DISTINCT genre FROM customer_staging WHERE genre IS NOT NULL;
```

Insert into `product` Table:

- **Context**: This insert statement populates the `product` table with distinct product descriptions and unit prices from a temporary table `online_retail` storing data from the 'online retail' uci dataset.. The data represents the unique all-occasion gifts sold by the online retail company.

Insert Statement:

```
INSERT INTO product (description, unit_price)
SELECT DISTINCT description, unitprice FROM online_retail;
```

Insert into `c_i` Table:

- **Context**: This insert statement populates the `c_i` table with random customer IDs and invoice numbers.

Insert Statement:

```
INSERT INTO c_i (customerid, invoiveno)
SELECT FLOOR(DBMS_RANDOM.VALUE(1, 200)) AS customer_id, invoiceno
FROM (SELECT DISTINCT invoiceno FROM online_retail);
```

Insert into `customer` Table:

- **Context**: This insert statement populates the `customer` table with randomly generated customer details, including first names, last names, gender IDs, age, annual income, spending score, and country IDs. The data is sourced from the `customer_staging` dataset.

Insert Statement:

```
INSERT INTO customer (first_name, last_name, gender_id, age,
annual_income_K$, spending_score, country_id)
SELECT
  DBMS_RANDOM.STRING('A', 10) AS first_name,
  DBMS_RANDOM.STRING('A', 10) AS last_name,
  g.gender_id,
  cs.age,
  cs.annual_income_K$,
  cs.spending_score,
  FLOOR(DBMS_RANDOM.VALUE(1, 32)) AS country_id
FROM customer_staging cs
INNER JOIN gender g ON g.gender = cs.genre;
```

Insert into `invoice` Table:

- **Context**: This insert statement populates the `invoice` table with invoice details, including invoice date, invoice number, customer ID, and total amount. The total amount is calculated based on the quantity and unit price of items in the `online_retail` dataset.

Insert Statement:

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

Insert into `invoice_line_item` Table:

- **Context**: This insert statement populates the `invoice_line_item` table with line item details, including invoice ID, product ID, quantity, and line total. It links the `online_retail` dataset to the `product` table.

Insert Statement:

```
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;
```

--*(corrected after the video, returns no error)*

These insert statements are used to populate the various tables in the database with data from the `online_retail` and `customer_staging` datasets. The data includes customer information, product details, and transaction records.

Read Operations: Select statements

Example:

Business Question: What is the total amount spent by each customer on all their purchases?

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

This query will group customers by their first and last names and sum up all the total amounts from the invoices related to each customer. It assumes that each customer_id is unique and first_name plus last_name will give a unique identification for a customer, which may not always be the case in practice.

Business Question: What are the most popular products sold, in terms of quantity?

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

This query will sum the quantities from the invoice lines for each product and order the results in descending order, showing the most popular products at the top.

Select statements we used in our project:

```
SELECT
  Gender,
  Age,
  Annual_Income,
  Spending_Score,
  Genre
FROM customer_staging;
```

- **Context**: Running the SQL query `select * from customer_staging` retrieves data from the `customer_staging` table. The `customer_staging` table serves as a temporary storage or staging area for customer information, specifically capturing details related to Gender, Age, Annual Income, and Spending Score. The `Genre` column represents gender information, and the `Annual Income` column is measured in thousands of dollars. The `Spending Score` can vary from 0 to 100. This query is used to access and analyze the raw customer data before it is processed and loaded into the main database. It allows for the examination of customer demographics and is useful for profiling, clustering analysis, or data cleansing purposes.

```
SELECT
  InvoiceNo,
  StockCode,
  Description,
  Quantity,
  InvoiceDate,
  UnitPrice,
  CustomerID,
  Country
FROM online_retail;
```

- **Context**: Executing the SQL query `select * from online_retail` retrieves data from the `online_retail` table. The `online_retail` table represents a transnational dataset containing all the transactions that occurred between December 1, 2010, and December 9, 2011, for a UK-based and registered non-store online retail company. The company primarily specializes in selling unique all-occasion gifts. Many of its customers are wholesalers. This dataset includes details about each transaction, such as Invoice Number, Product Code, Product Name, Quantity, Invoice Date, Unit Price, Customer Number, and Country Name. The data in this table is essential for tracking and analyzing customer transactions, product sales, and customer locations, providing valuable insights into the company's operations and customer behavior.

```
SELECT
  c.customer_id,
  c.first_name,
  c.last_name,
  c.age,
  c.annual_income_K$,
  c.spending_score,
  ctr.country
FROM customer c
INNER JOIN country ctr
ON c.country_id = ctr.country_id;
```

- **Context**: The following SQL SELECT statement is used to retrieve and associate customer data with their respective countries. It leverages an INNER JOIN to combine information from the `customer` table and the `country` table. The `customer` table contains details about the customers, such as their first name, last name, age, annual income, spending score, and a reference to their country through the `country_id`. The `country` table holds information about the countries, and it includes a unique identifier for each country (`country_id`) and the country name (`country`).

Update operations:

Examples of update statements

Bussiness Question:

A customer wants to take more pieces of an item than described in the invoice and they want you to make sure this is refeltected in the invoice line for that item rather than adding a new line to the invoice. Invoice_no = '536365', item described as 'WHITE HANGING HEART T-LIGHT HOLDER'

```
 update invoive_line_item set quantity = 6  where invoice_id = (select
invoice_id from invoice where invoice_no = '536365') and product_id =
(select from product where decription = 'WHITE HANGING HEART T-LIGHT
HOLDER')
```

Update Statements used in our development:

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

Context : The SQL UPDATE statement is used to modify existing data in the `online_retail` dataset. In this scenario, the update is focused on the `customerid` column. The goal is to associate each transaction (invoice) in the dataset with the respective customer by matching the `invoiceno` with the corresponding customer information stored in the `c_i` table. This update operation aims to link each transaction to the customer data in the 'customer data' dataset.By executing this UPDATE statement, the `customerid` column in the `online_retail` dataset will be populated with the appropriate customer identifiers from the second dataset will be joined using this update.

Delete Statements:

Example for deleting selective data:

Bussiness Question: A customer changes his mind on an item while billing and wants to remove an item described as ‘WHITE HANGING HEART T-LIGHT HOLDER’ from his invoice with invoice number '536365’.

```
Delete from invoive_line_item where invoice_id = (select invoice_id from
invoice where invoice_no = ‘536365’) and product_id = (select from product
where decription = ‘WHITE HANGING HEART T-LIGHT HOLDER’)
```

Delete all Data:

DELETE FROM "country" Table:

- **Context:** This operation removes all data from the "country" table.
- **Statement:**

```
DELETE FROM country;
```

DELETE FROM "gender" Table:

- **Context:** This operation removes all data from the "gender" table.
- **Statement:**

```
DELETE FROM gender;
```

DELETE FROM "product" Table:

- **Context:** This operation removes all data from the "product" table.
- **Statement:**

```
DELETE FROM product;
```

DELETE FROM "customer" Table:

- **Context:** This operation removes all data from the "customer" table.
- **Statement:**

```
DELETE FROM customer;
```

DELETE FROM "invoice" Table:

- **Context:** This operation removes all data from the "invoice" table.
- **Statement:**

```
DELETE FROM invoice;
```

DELETE FROM "invoice_line_item" Table:

- **Context:** This operation removes all data from the "invoice_line_item" table.
- **Statement:**

```
DELETE FROM invoice_line_item;
```

Drop Statements:

DROP for `country` Table:

- **Context**: This operation removes the `country` table and its associated elements, including the sequence, trigger, and foreign key constraint.
- **Statements**:

```
DROP SEQUENCE country_id_sequence;

DROP TRIGGER country_id_trigger;

ALTER TABLE customer DROP CONSTRAINT fk_country_id;

DROP TABLE country;
```

DELETE FROM and DROP for `gender` Table:

- **Context**: This operation removes the `gender` table and its associated elements, including the sequence, trigger, and foreign key constraint.
- **Statements**:

```
DROP SEQUENCE gender_id_sequence;

DROP TRIGGER gender_id_trigger;

ALTER TABLE customer DROP CONSTRAINT fk_gender_id;

DROP TABLE gender;
```

DELETE FROM and DROP for `product` Table:

- **Context**: This operation removes the `product` table and its associated elements, including the sequence and trigger. Additionally, it drops a foreign key constraint from the `invoice_line_item` table.
- **Statements**:

```
DROP SEQUENCE product_id_sequence;

DROP TRIGGER product_id_trigger;

ALTER TABLE invoice_line_item DROP CONSTRAINT fk_product_id;

DROP TABLE product;
```

DROP for `customer` Table:

- **Context**: This operation removes the `customer` table and its associated elements, including the sequence, trigger, and foreign key constraint.
- **Statements**:

```
DROP SEQUENCE customer_id_sequence;

DROP TRIGGER customer_id_trigger;

ALTER TABLE invoice DROP CONSTRAINT fk_customer_id;

DROP TABLE customer;
```

DROP for `invoice` Table:

- **Context**: This operation removes the `invoice` table and its associated elements, including the sequence and trigger. It also drops a foreign key constraint from the `invoice_line_item` table.
- **Statements**:

```
DROP SEQUENCE invoice_id_sequence;

DROP TRIGGER invoice_id_trigger;

ALTER TABLE invoice_line_item DROP CONSTRAINT fk_invoice_id;

DROP TABLE invoice;
```

DROP INDEX for `online_retail` Table:

- **Context**: This operation removes the indexes created on the `invoiceno` and `description` columns in the `online_retail` dataset.
- **Statements**:

```
DROP INDEX invoiceno_index;

DROP INDEX description_index;
```

DROP for `invoice_line_item` Table:

- **Context**: This operation removes the `invoice_line_item` table and its associated elements, including the sequence and trigger.
- **Statements**:

```
DROP SEQUENCE invoice_line_id_sequence;

DROP TRIGGER invoice_line_id_trigger;

DROP TABLE invoice_line_item;
```

These SQL statements and operations are used to remove the tables, sequences, triggers, constraints, and indexes associated with the database schema. The context provided describes the purpose and impact of each operation.

ETL: A step by step recollection of my Data Importation Process

Importing Data into Oracle Using SQL Developer:

Step 1: Launch SQL Developer

- Open SQL Developer on your laptop.
- Connect to the Oracle database.

Step 2: Navigate to Tables

- In SQL Developer, go to the "Tables" section.

Step 3: Initiate Data Import

- Right-click on the "Tables" section.
- Choose "Import Data" from the context menu.

Step 4: Specify Source Data

- Select the source data type (e.g., CSV file).
- Leave the "File Name" field blank for later input.

Step 5: Define Target Table

- Choose an existing target table or create a new one.
- Define the target table structure, if necessary.

Step 6: Verify Column Mappings

- Review automatic column mappings.
- Adjust mappings if needed to match source and target columns.

Step 7: Configure Import Options

- Customize import options (e.g., handle duplicates, errors, constraints).

Step 8: Data Preview

- Preview the data to ensure accuracy.

Step 9: Start the Import

- Click "Next" or "Finish" to initiate the import process.

Step 10: Monitor Progress

- Monitor the import progress within the wizard.

Step 11: Review Import Results

- Check the summary of import results (imported rows, errors).

Step 12: Verify Imported Data

- Validate the imported data in the target table using SQL queries and the SQL Developer interface.

Conclusion:

- The data import process was efficient and allowed for seamless data transfer into the Oracle database.

Queries used in the video:

```
select * from online_retail


select * from customer_staging



create table country (

country_id int primary key,

country varchar(50)

);


CREATE SEQUENCE country_id_sequence

START WITH 1

INCREMENT BY 1

NOCACHE

NOCYCLE;


CREATE OR REPLACE TRIGGER country_id_trigger

BEFORE INSERT ON country

FOR EACH ROW

BEGIN

  SELECT country_id_sequence.NEXTVAL

  INTO :new.country_id

  FROM dual;

END;
```

```
/




create table gender (
gender_id int primary key,
gender varchar(50)
);




CREATE SEQUENCE gender_id_sequence
START WITH 1
INCREMENT BY 1
NOCACHE
NOCYCLE;

CREATE OR REPLACE TRIGGER gender_id_trigger
BEFORE INSERT ON gender
FOR EACH ROW
BEGIN
  SELECT gender_id_sequence.NEXTVAL
  INTO :new.gender_id
  FROM dual;
END;
/

create table product (
product_id int primary key,
description varchar(1000),
unit_price numeric(10,5)
);
```

```
CREATE SEQUENCE product_id_sequence
START WITH 1
INCREMENT BY 1
NOCACHE
NOCYCLE;

CREATE OR REPLACE TRIGGER product_id_trigger
BEFORE INSERT ON product
FOR EACH ROW
BEGIN
  SELECT product_id_sequence.NEXTVAL
  INTO :new.product_id
  FROM dual;
END;
/


create table customer (
customer_id int 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) );
```

```
CREATE SEQUENCE customer_id_sequence
START WITH 1
INCREMENT BY 1
NOCACHE
NOCYCLE;


CREATE OR REPLACE TRIGGER customer_id_trigger
BEFORE INSERT ON customer
FOR EACH ROW
BEGIN
  SELECT customer_id_sequence.NEXTVAL
  INTO :new.customer_id
  FROM dual;
END;
/

create table invoice (
invoice_id int 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)


);

CREATE SEQUENCE invoice_id_sequence
START WITH 1
INCREMENT BY 1
```

```
NOCACHE
NOCYCLE;

CREATE OR REPLACE TRIGGER invoice_id_trigger
BEFORE INSERT ON invoice
FOR EACH ROW
BEGIN
  SELECT invoice_id_sequence.NEXTVAL
  INTO :new.invoice_id
  FROM dual;
END;
/



create table invoice_line_item (
invoice_line_id int 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)

);

CREATE SEQUENCE invoice_line_id_sequence
START WITH 1
INCREMENT BY 1
NOCACHE
NOCYCLE;
```

```
CREATE OR REPLACE TRIGGER invoice_line_id_trigger
BEFORE INSERT ON invoice_line_item
FOR EACH ROW
BEGIN
  SELECT invoice_line_id_sequence.NEXTVAL
  INTO :new.invoice_line_id
  FROM dual;
END;
/



CREATE INDEX invoiceno_index
ON online_retail (invoiceno);

CREATE INDEX description_index
ON online_retail (description);



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



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

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



BEGIN
   DBMS_RANDOM.seed(42);

END;
```

```
/


create table c_i(
customerid int,
invoiveno int );


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 )


INSERT INTO customer (first_name, last_name, gender_id, age,
annual_income_K$, spending_score, country_id)
SELECT
  DBMS_RANDOM.STRING('A', 10) AS first_name,
  DBMS_RANDOM.STRING('A', 10) AS last_name,
  g.gender_id,
  cs.age,
  cs.annual_income_K$,
  cs.spending_score,
  FLOOR(DBMS_RANDOM.VALUE(1, 32)) AS country_id
FROM customer_staging cs
INNER JOIN gender g ON g.gender = cs.genre;


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


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
```