Data Engineering

SQL

SQL Keywords and Their Descriptions

Below is a list of common SQL keywords used in queries, along with their descriptions:

- `SELECT`: Used to select data from a database.
- `FROM`: Specifies the table from which to select or delete data.
- `WHERE`: Adds a condition to the SQL query.
- `INSERT INTO`: Used to insert new data into a database.
- `UPDATE`: Updates existing data within a table.
- `DELETE`: Deletes existing data from a table.
- `CREATE TABLE`: Creates a new table in the database.
- `ALTER TABLE`: Modifies an existing table structure, such as adding or deleting columns.
- `DROP TABLE`: Deletes an entire table from the database.
- `JOIN`: Combines rows from two or more tables, based on a related column between them.
- `GROUP BY`: Groups rows that have the same values in specified columns into summary rows.
- `ORDER BY`: Sorts the result set in ascending or descending order.
- `HAVING`: Adds a condition to the `GROUP BY` clause, filtering the groups to include in the results.
- `LIMIT`: Specifies the maximum number of records to return in the result set.
- `DISTINCT`: Returns only distinct (different) values in the result set.

These keywords are the building blocks of SQL queries, allowing for the manipulation and retrieval of data stored in relational databases.

Types of Filtering Operations in SQL

Filtering operations in SQL are primarily used to narrow down the result set by specifying conditions that the data must meet. Here are the main types of filtering operations that can be performed on an SQL database:

- `WHERE`: Allows for the specification of conditions to filter rows in a single table. It can be used with operators such as `=`, `<>`, `>`, `<`, `>=`, `<=`, `BETWEEN`, `LIKE`, and `IN`.
- `HAVING`: Similar to the `WHERE` clause, but used to filter rows after an aggregation has been performed. It is typically used in conjunction with the `GROUP BY` clause to filter groups or aggregates.

- `LIKE`: Used within a `WHERE` clause to search for a specified pattern in a column.
- `IN`: Allows you to specify multiple values in a `WHERE` clause, effectively filtering rows based on a range of values in a column.
- `BETWEEN`: Enables the selection of rows within a certain range. It can be used with numbers, text, and dates.
- `IS NULL` / `IS NOT NULL`: Used in a `WHERE` clause to filter rows based on whether a column's value is null or not.
- `EXISTS`: Used to test for the existence of any record in a subquery, returning true if the subquery returns one or more records.

Views:

A view in SQL is a virtual table based on the result-set of an SQL statement. It contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database. You can use views to:

- Simplify complex queries by encapsulating them into a view.
- Restrict access to specific data by creating views that contain only the necessary information.
- Present data in a particular format or perspective.

Creating a View:
To create a view, you use the `CREATE VIEW` statement. Here's the basic syntax:

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Accessing Views in SQL:
Accessing data from a view in SQL is similar to accessing data from a real table. You can use the `SELECT` statement to query data from a view. The syntax for accessing data from a view is the same as accessing data from a table.

```
SELECT column1, column2, ...
FROM view_name
WHERE condition;
```

Sorting in SQL:
Sorting in SQL is performed using the `ORDER BY` clause. This clause is used to sort the result set of a query by one or more columns. It can sort the data in ascending order (using the `ASC` keyword) or descending order (using the `DESC` keyword). If no keyword is specified, the default sort order is ascending.

```sql
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...;
```

Grouping in SQL:
Grouping in SQL is primarily achieved using the `GROUP BY` clause. This clause is used in collaboration with SQL aggregate functions (like `COUNT`, `MAX`, `MIN`, `SUM`, `AVG`) to group the result set by one or more columns. Grouping is essential for performing aggregate calculations on each group.

```sql
SELECT column1, aggregate_function(column2)
FROM table_name
WHERE condition
GROUP BY column1;
```

Joins

1. **Inner Join in SQL**
   The `INNER JOIN` keyword in SQL is used to combine rows from two or more tables, based on a related column between them. It selects rows that have matching values in both tables, providing a way to query across multiple tables as if they were a single table.

```sql
SELECT table1.column1, table2.column2, ...
FROM table1
INNER JOIN table2
ON table1.common_column = table2.common_column;
```

2. **Outer Join in SQL**
   An Outer Join in SQL is used to return a set of records that includes what is in the left table, the right table, or both. It is divided into three types: LEFT JOIN (or LEFT OUTER JOIN), RIGHT JOIN (or RIGHT OUTER JOIN), and FULL JOIN (or FULL OUTER JOIN). These joins are crucial for querying data from multiple tables to find records with matching values in their common columns, and also to find records with no matches.
3. **Right Outer JOIN**
   A **Right Outer Join** in SQL is a type of join that returns all rows from the right (second) table, and the matched rows from the left (first) table. If there is no match, the result set will include NULL on the side of the left table.

```sql
SELECT columns
FROM table1
RIGHT OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

4. **Left Outer Join**
   The LEFT JOIN returns all records from the left table, and the matched records from the right table. If there is no match, the result is NULL on the side of the right table.

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.common_column = table2.common_column;
```

5. **Full Outer Join:**
   A `FULL OUTER JOIN` in SQL is a type of join that returns all rows from both tables involved in the join, regardless of whether a matching row exists in the other table. If there is a match between the columns being joined, the `FULL OUTER JOIN` combines the matched rows into a single row. For rows in one table that do not have matching rows in the other table, the result set will still include these rows, but with `NULL` values for each column from the table where a matching row does not exist.

```
SELECT columns
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

6. **Cross Join:**
   A **Cross Join** in SQL, also known as a Cartesian Join, is a join operation that pairs each row from two tables, resulting in a Cartesian product of the two tables. This means if the first table has `N` rows and the second table has `M` rows, the result set will have `N * M` rows, assuming there are no filters applied to the join.

```
SELECT *
FROM TableA
CROSS JOIN TableB;
```

7. **Self Join:**
   A Self Join in SQL is a join in which a table is joined with itself. This type of join is used when you need to compare rows within the same table. For example, you might use a self join to find pairs of records that meet certain criteria or to compare rows within the same table to find duplicates.

```
SELECT A.column_name, B.column_name
FROM table_name AS A, table_name AS B
WHERE A.common_field = B.common_field
AND A.unique_field <> B.unique_field;
```

8. **Chain of Joins in SQL**
   Chaining joins in SQL allows you to retrieve data from multiple tables by performing multiple join operations in a single query. This is particularly useful when the data you need is distributed across several tables, and you need to combine them based on related columns. You can chain `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, and `FULL JOIN` depending on your requirements.

```
SELECT table1.column1, table2.column2, table3.column3
FROM table1
INNER JOIN table2 ON table1.common_column = table2.common_column
INNER JOIN table3 ON table2.another_common_column =
table3.another_common_column;
```

Set Theory

1. **UNION:**
   The `UNION` operator in SQL is used to combine the result sets of two or more `SELECT` statements. It removes duplicate rows between the various `SELECT` statements. Each `SELECT` statement within the `UNION` must have the same number of columns in the result sets with similar data types. The columns in each `SELECT` statement must also be in the same order.

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

2. **UNION ALL:**
   The `UNION ALL` operator in SQL is used to combine the result sets of two or more `SELECT` statements, including all duplicates. Unlike the `UNION` operator, which removes duplicate rows between the various `SELECT` statements, `UNION ALL` will include every row from the result sets. This can be particularly useful when you need to maintain duplicate rows for the combined dataset. Each `SELECT` statement within the `UNION ALL` must have the same number of columns in the result sets with similar data types, and the columns must be in the same order.

```
SELECT column_name(s) FROM table1
UNION ALL
SELECT column_name(s) FROM table2;
```

3. **INTERSECT:**
   The `INTERSECT` operator in SQL is used to return the intersection of two or more `SELECT` statements. It means that it will return only the rows that are available in both `SELECT` statement result sets. This is useful when you need to find common elements

between different datasets. Like `UNION` and `UNION ALL`, each `SELECT` statement within the `INTERSECT` must have the same number of columns, with the columns having similar data types and in the same order.

```
SELECT column_name(s) FROM table1
INTERSECT
SELECT column_name(s) FROM table2;
```

4. **EXCEPT:**
   The `EXCEPT` operator in SQL is used to return all rows from the first `SELECT` statement that are not present in the second `SELECT` statement. Essentially, it performs a set difference operation between two datasets. The rows returned by the `EXCEPT` operation are distinct, meaning it removes duplicates. Each `SELECT` statement within the `EXCEPT` must have the same number of columns, and those columns must have similar data types. Additionally, the columns must be in the same order in each `SELECT` statement.

```
SELECT column_name(s) FROM table1
EXCEPT
SELECT column_name(s) FROM table2;
```

Set vs Joins

1. **Purpose**:
   - **Set Theory Operations**: Aimed at handling sets of data to perform union, intersection, and set difference operations, focusing on the uniqueness and membership of elements.
   - **Joins**: Aimed at combining data from two or more tables based on a related column, focusing on relational data retrieval.
2. **Operation**:
   - **Set Theory Operations**: Do not require a relationship between datasets other than the elements being compared. They work on the principle of set membership.
   - **Joins**: Require a defined relationship (usually a foreign key relationship) between the tables being joined. They work on matching column values.
3. **Resultant Data Structure**:
   - **Set Theory Operations**: The result is a set that contains unique elements, adhering to the rules of set theory (e.g., no duplicate elements in the case of union).
   - **Joins**: The result is a table that combines columns from the joined tables, which can include duplicate rows based on the join condition.
4. **Use Cases**:
   - **Set Theory Operations**: Useful for analysing the presence or absence of data across different datasets, such as finding common elements (intersection) or

differences.

- **Joins**: Essential for relational database operations where data is normalised and stored across multiple tables but needs to be viewed or analysed together.

Sub Querying

1. **Semi Joins**
   A SEMI JOIN in SQL is a type of join that is used to return all rows from the first table (left table) where at least one row exists in the second table (right table) that meets the join condition. However, unlike other joins, SEMI JOIN does not return any columns from the right table; it only checks for the existence of matching rows. This makes SEMI JOIN particularly useful for filtering rows based on the existence of related data in another table.

```
SELECT columns
FROM table1
WHERE column IN (SELECT column FROM table2 WHERE condition);
```

2. **Anti Joins**
   An ANTI JOIN in SQL is used to return all rows from the first table (left table) where no rows exist in the second table (right table) that meet the join condition. Essentially, it filters out the rows that have matching entries in another table, returning only those that do not have a corresponding match. This is useful for finding records in one table that have no related records in another table.

```
SELECT columns
FROM table1
WHERE column NOT IN (SELECT column FROM table2 WHERE condition);
```

RDBMS

CREATE TABLE

To create a table in SQL, you use the `CREATE TABLE` statement. This statement allows you to define the table's structure, including its columns and their data types, as well as any constraints (like primary keys) that you want to apply to ensure data integrity. Here's the basic syntax:

```
CREATE TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
    column3 datatype constraint,
    ...
);
```

ALTER TABLE

The `ALTER TABLE` statement in SQL is used to modify the structure of an existing table in a database. This can include adding, deleting, or modifying columns, as well as changing the data type of a column or adding constraints. Here's the basic syntax for various operations you can perform with `ALTER TABLE`:

1. **Add a Column**

```
ALTER TABLE table_name
ADD column_name datatype;
```

2. **Rename a Column**

```
ALTER TABLE table_name
RENAME COLUMN old_column_name TO new_column_name;
```

3. **Modify a Column Data Type**

```
ALTER TABLE table_name
MODIFY COLUMN column_name datatype;
```

4. **Drop a Column**

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

5. **DROP TABLE**

```
DROP TABLE table_name;
```

6. **RENAME TABLE**

```
RENAME TABLE old_table_name TO new_table_name;
```

Constraints

1. PRIMARY KEY: Ensures that all values in a column are unique and not NULL.
2. FOREIGN KEY: Ensures that the values in a column or a group of columns match the values in a column of another table.
3. UNIQUE: Guarantees that all values in a column are different.
4. NOT NULL: Specifies that a column cannot hold a NULL value.
5. CHECK: Ensures that all values in a column satisfy a specific condition.

6. DEFAULT: Sets a default value for a column when no value is specified.
7. INDEX: Used to create and retrieve data from the database very quickly.

Domain Constraints: Refers to the Data type to the column or attribute.
Referial Integrity: Violation occurs when foreign key doesn't exist in the primary table which it is referred from

NOTE:

1. Casting in SQL refers to the process of converting a value from one data type to another. This is particularly useful when you need to perform operations between columns of different data types or when you need to format output data in a specific way. SQL provides the `CAST` function to perform explicit data type conversion according to the needs of a query.

   Syntax:

```
CAST(expression AS target_type)
```

2. To add a unique constraint to an existing table column, you use the ALTER TABLE statement. Here is the basic

   Syntax:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name UNIQUE (column_name);
```

Keys

1. **Primary Key:**
   A **Primary Key** in SQL is a column or a combination of columns that uniquely identifies each row in a table. It ensures that no duplicate values are entered in the column(s) it is defined on, and it cannot accept `NULL` values. This makes it an essential part of relational database design, ensuring data integrity and enabling efficient data retrieval.

```
CREATE TABLE Students (
    StudentID int NOT NULL,
    FirstName varchar(255),
    LastName varchar(255),
    Age int,
    PRIMARY KEY (StudentID)
);
```

2. **Surrogate Key:**
   A **Surrogate Key** in SQL is a type of primary key that is artificially generated by the database. It is not derived from application data. Surrogate keys are typically used when a natural primary key (one that is meaningful to the user) is either not available or not desirable. They are usually implemented as an auto-incrementing number or a globally unique identifier (GUID). Surrogate keys have several advantages:

- **Uniqueness**: They ensure each row in a table can be uniquely identified.
- **Simplicity**: They are simple to understand and use, especially when dealing with complex data structures.
- **Performance**: They can improve the performance of database operations, as they are typically small, fixed-length fields that are indexed efficiently.
- **Stability**: Their values do not change, which helps maintain referential integrity across tables.

```
CREATE TABLE Orders (
    OrderID int NOT NULL AUTO_INCREMENT,
    OrderDate date NOT NULL,
    CustomerID int,
    Amount decimal(10,2),
    PRIMARY KEY (OrderID)
);
```

3. **Foreign key**
   A **foreign key** in SQL is a key used to link two tables together. It is a field (or collection of fields) in one table that uniquely identifies a row of another table or the same table. In simple terms, a foreign key is a column or a combination of columns that is used to establish and enforce a link between the data in two tables.

   To initialise a foreign key in SQL, you use the `FOREIGN KEY` constraint when you create or alter a table. This constraint ensures that the relationship between the two tables remains consistent. When a `FOREIGN KEY` constraint is in place, it prevents actions that would destroy links between tables. Additionally, it ensures that the foreign key columns provide only values that match the primary key in the referenced table, or null.

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    CustomerID int,
    OrderDate date NOT NULL,
    PRIMARY KEY (OrderID),
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

DBMS

OLTP vs OLAP

Feature	OLTP (Online Transaction Processing)	OLAP (Online Analytical Processing)
Primary Focus	Managing transaction-oriented applications.	Analysis of business measures by categories and attributes.
Data Updates	Frequent inserts, updates, and deletes.	Bulk data modification through periodic updates.
Database Design	Normalized tables to avoid data redundancy.	Denormalized tables to improve query performance.
Query Types	Simple, standard transactions (INSERT, UPDATE, DELETE).	Complex queries involving aggregations (SUM, COUNT, AVG) and joins over large datasets.
Transaction Volume	High volume of short online transactions.	Low volume of transactions but complex queries that are resource-intensive.
Response Time	Milliseconds	Seconds to minutes, depending on the complexity and data volume.
User Types	Clerks, DBAs, or database professionals managing day-to-day operations.	Analysts, managers, or decision-makers looking for insights into business performance, trends, and forecasting.
Example Use Cases	Order entry, retail sales, financial transactions.	Business reporting, data mining, complex analytical calculations, and decision support systems.
Data Consistency	Immediate consistency is crucial for transaction accuracy.	Consistency is important but can tolerate some latency due to the nature of batch updates.
System Architecture	Designed for OLTP databases to handle many small transactions quickly.	Designed for OLAP databases to process large volumes of data and complex queries efficiently.
Storage Requirements	Generally lower than OLAP because of normalization and focus on current data.	Higher due to denormalization, historical data, and aggregated data storage needs.
Indexing Strategies	Primary keys and foreign keys are heavily used for transaction speed.	Bitmap indexes and materialized views are common to enhance query performance on large datasets.

Realtion Database Model

1. **Physical Database Model:**
    - This model describes how data is stored in the database, including the physical storage devices and access paths. It details the actual storage mechanism, file formats, indexing techniques, and so forth. The physical model is concerned with the performance and efficiency of the database system.
2. **Logical Database Model:**
    - The logical model defines the structure of the data as it appears to the end-user, without getting into the details of physical storage. It includes entities, attributes, and relationships among them. This model is more about the schema and the design of the database rather than how data is stored or accessed. It serves as a bridge between the conceptual model and the physical model.
3. **Conceptual Database Model:**
    - This model provides a high-level view of the database without getting into the details of how data is stored or structured in the database. It focuses on what data is stored in the database and the relationships among those data entities. The conceptual model is useful for understanding the overall structure of the database at a very abstract level and is often used in the initial design phase.

Dimensional Model Design

The Dimensional Model is a design technique used primarily for data warehousing and business intelligence. It organises data into two types of tables: facts and dimensions. This model is designed to improve data readability and query performance in a database. Here's a brief overview of its components:

1. **Fact Tables:**
    - Fact tables store quantitative data for analysis and reporting. These tables contain metrics, measurements, or facts of a business process (e.g., sales revenue, quantity sold). Fact tables are typically large and contain foreign keys that uniquely identify related dimension table rows.
2. **Dimension Tables:**
    - Dimension tables contain descriptive attributes related to the dimensions of a business process. These attributes are textual fields (e.g., product name, date, employee name) and are used to categorize, filter, or label facts. Dimension tables are usually smaller than fact tables and provide context to the data in the fact tables.
3. **Star Schema:**
    - One common architecture of the dimensional model is the Star Schema, where a central fact table is directly connected to multiple dimension tables. This schema resembles a star, with the fact table at the center and dimension tables radiating outwards. It simplifies queries and enhances data retrieval efficiency.
4. **Snowflake Schema:**

    - The Snowflake Schema is a variation of the Star Schema where dimension tables are normalised into multiple related tables. This reduces data redundancy but can make queries more complex due to the additional joins required.
5. **Benefits of Dimensional Model:**
    - The dimensional model is highly optimised for data warehousing and analytical processing. It offers several benefits, including simplified queries, improved data comprehension, and faster data retrieval. This model supports ad-hoc reporting and can handle large volumes of data efficiently.

Normalisation

Normalisation in SQL is a database design technique that reduces data redundancy and improves data integrity. It involves organising the fields and tables of a database to minimize redundancy and dependency by dividing large tables into smaller, less redundant tables and defining relationships between them. The ultimate goal of normalisation is to isolate data so that additions, deletions, and modifications can be made in just one table and then propagated through the rest of the database via the defined relationships.

There are several normal forms, each with more strict rules than the previous one. The most commonly used normal forms are:

1. **First Normal Form (1NF):** This requires that the values in each column of a table are atomic, meaning each column must have a unique value for each row of data. There should be no repeating groups or arrays.
2. **Second Normal Form (2NF):** For a table to be in the second normal form, it must first satisfy all the conditions of the first normal form. In addition, it should have no partial dependency; that is, no column value should depend on only a part of a primary key.
3. **Third Normal Form (3NF):** A table is in the third normal form if it is in the second normal form and all of its columns are not transitively dependent on the primary key. In simpler terms, no non-primary key attribute should depend on another non-primary key attribute.
4. **Boyce-Codd Normal Form (BCNF):** Sometimes considered a stricter version of the third normal form, a table is in BCNF if it is in 3NF and every determinant is a candidate key. This form deals with certain types of anomaly not handled by 3NF.
5. **Fourth Normal Form (4NF):** A table is in 4NF if it is in the Third Normal Form and it does not have any multi-valued dependencies. This means that there should be no two or more independent multi-valued facts about the same entity.
6. **Fifth Normal Form (5NF):** Also known as Project-Join Normal Form (PJNF), a table is in 5NF if it is in 4NF and every join dependency in the table is a consequence of the candidate keys.

Permissions

In SQL, these Data Control Language (DCL) statements manage access privileges for users and roles within a database.

1. **GRANT:**
   Assigns specific permissions to users or roles on database objects (tables, views, procedures, etc.).

   Syntax:

```
GRANT privilege_name [, privilege_name]...
ON object_name [, object_name]...
TO { user_name | PUBLIC | role_name }
[ WITH GRANT OPTION ]
```

1. `privilege_name`: The permission being granted (e.g., SELECT, INSERT, UPDATE, DELETE).
2. `object_name`: The database object on which the permission is granted.
3. `user_name`: The user to whom the permission is granted.
4. `PUBLIC`: Grants access to all users.
5. `role_name`: Grants access to a specific role.
6. `WITH GRANT OPTION`: Allows the grantee to further grant the same permission to others (depends on database system).
7. **REVOKE:**
   Removes previously granted permissions from users or roles.
   Syntax:

```
GRANT SELECT, INSERT ON my_table TO user1; -- Grant SELECT and INSERT
to user1 on my_table
```

REVOKE UPDATE ON my_table FROM user1; -- Revoke UPDATE permission from user1 on my_table

```
   **Example:**
```sql
REVOKE privilege_name [, privilege_name]...
ON object_name [, object_name]...
FROM { user_name | PUBLIC | role_name }
```

Group and User Roles
```

1. **User:** An individual database user with a unique username and password. Permissions are typically granted directly to users.
2. **Group (Role):** A named collection of permissions that can be assigned to multiple users. Roles promote efficient permission management:
3. Grant permissions to a role.
4. Assign users to the role.
5. Changes to role permissions automatically affect all users in the role.

Example:

1. Create a role named `sales_team` with `SELECT` and `INSERT` permissions on the `customers` table.
2. Assign users `user2` and `user3` to the `sales_team` role.

```
CREATE ROLE sales_team;

GRANT SELECT, INSERT ON customers TO sales_team;

ALTER ROLE sales_team ADD MEMBER user2;
ALTER ROLE sales_team ADD MEMBER user3;
```

Now, both `user2` and `user3` inherit the permissions granted to the `sales_team` role.

By effectively using GRANT, REVOKE, Groups, and User roles, you can establish a robust and secure permission system for your database, ensuring users only have the access they need.

Examples:

```
GRANT {group_role} TO {user};
REVOKE {group_role} FROM {user};
```

Vertical partitioning

Vertical partitioning in SQL involves dividing a table into smaller tables where each new table contains a subset of the columns from the original table. This approach is used to improve performance, manageability, and scalability by segregating data based on column access patterns. Each partitioned table holds the same number of rows but fewer columns. A common use case is separating frequently accessed columns from infrequently accessed ones.

Unfortunately, SQL itself does not have a built-in syntax specifically for vertical partitioning as this concept is more related to database design and implementation rather than a direct SQL command. However, implementing vertical partitioning typically involves creating new

tables and then distributing the columns of the original table among them. Here's a conceptual approach to how you might manually implement vertical partitioning:

1. **Identify Columns for Partitioning**: Determine which columns are accessed together frequently and group them.
2. **Create New Tables**: For each group of columns identified, create a new table. Ensure each new table includes the primary key of the original table to maintain relationships.
3. **Migrate Data**: Copy the data from the original table into the new tables, ensuring that each new table contains the appropriate subset of columns.
4. **Update Application Logic**: Modify your application logic to access the new tables instead of the original table.

Example:
Suppose we have a table `UserProfile` with columns `UserID`, `UserName`, `Email`, `Address`, and `LastLoginDate`. We decide to vertically partition it into two tables: one for frequently accessed user information and another for login details.

```
-- Original table
CREATE TABLE UserProfile (
    UserID INT PRIMARY KEY,
    UserName VARCHAR(100),
    Email VARCHAR(100),
    Address VARCHAR(200),
    LastLoginDate DATETIME
);

-- Table 1: User information
CREATE TABLE UserInfo (
    UserID INT PRIMARY KEY,
    UserName VARCHAR(100),
    Email VARCHAR(100),
    Address VARCHAR(200)
);

-- Table 2: User login details
CREATE TABLE UserLogin (
    UserID INT PRIMARY KEY,
    LastLoginDate DATETIME
);

-- Migrate data to new tables (example for UserInfo)
INSERT INTO UserInfo (UserID, UserName, Email, Address)
SELECT UserID, UserName, Email, Address FROM UserProfile;

-- Migrate data to new tables (example for UserLogin)
INSERT INTO UserLogin (UserID, LastLoginDate)
SELECT UserID, LastLoginDate FROM UserProfile;
```

Horizontal partitioning

Horizontal partitioning in SQL is a database design principle used to improve performance and manageability by dividing a table into smaller, more manageable pieces called partitions. Each partition contains rows that are logically related, typically based on a range or list of values in one or more columns. This approach can significantly enhance query performance, especially for large tables, by allowing operations to target only relevant partitions.

Syntax:
Horizontal partitioning can be implemented in various SQL databases using different syntaxes. Below is a general example using the `CREATE TABLE` statement with partitioning clauses, which might vary depending on the specific SQL database management system (DBMS) you are using.

```sql
CREATE TABLE orders (
    order_id INT,
    order_date DATE,
    customer_id INT,
    amount DECIMAL(10,2)
) PARTITION BY RANGE (order_date) (
    PARTITION p0 VALUES LESS THAN ('2020-01-01'),
    PARTITION p1 VALUES LESS THAN ('2021-01-01'),
    PARTITION p2 VALUES LESS THAN ('2022-01-01'),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

Python for DE

Import Excel Sheets using Pandas

1. **Load an Excel file into a pandas DataFrame**

```python
excel_file = 'path_to_your_excel_file.xlsx'
df = pd.read_excel(excel_file)
```

2. **If the Excel file has multiple sheets and you want to specify a particular sheet**

```python
sheet_name = 'your_sheet_name'
df_specific_sheet = pd.read_excel(excel_file, sheet_name=sheet_name)
```

3. **To get a list of all sheet names in the Excel file**

```
xls = pd.ExcelFile(excel_file)
sheet_names = xls.sheet_names
print(sheet_names)
```

Import Flat Files using NumPy

NumPy provides several functions to load data from text files. Each of these functions serves different purposes and comes with its own set of parameters to handle various data loading scenarios.

1. `loadtxt`
    - **Purpose**: Used to load data from a text file, where each row in the text file must have the same number of values.
    - **Usage**: `numpy.loadtxt(fname, dtype=float, delimiter=None, ...)`
    - **Parameters**:
        1. `fname`: File, filename, or generator to read.
        2. `dtype`: Data type of the resulting array; default is float.
        3. `delimiter`: String to separate values; default is white space.
    - **Advantages**: Simple and easy to use for loading and parsing regular, well-formed data files.
    - **Limitations**: Not as flexible as `genfromtxt` when dealing with missing values or variable-length rows.
2. `genfromtxt`
    - **Purpose**: More flexible than `loadtxt`, designed to handle missing values and variable-length rows.
    - **Usage**: `numpy.genfromtxt(fname, dtype=float, delimiter=None, ...)`
    - **Parameters**:
        1. `fname`: File, filename, or generator to read.
        2. `dtype`: Data type of the resulting array; can specify different types for different columns.
        3. `delimiter`: String to separate values; default is white space.
        4. `filling_values`: Specifies the value to use for missing values.

    - **Advantages**: Highly flexible, can handle datasets that are not well-formed, supports missing values and variable-length rows.
    - **Limitations**: Slightly more complex and slower than `loadtxt` due to its additional flexibility.

3. `recfromcsv`
    - **Purpose**: A convenience function that assumes the data is in CSV format and uses `genfromtxt` under the hood with default parameters suitable for common CSV files.

- **Usage**: `numpy.recfromcsv(fname, dtype=None, delimiter=',', ...)`
- **Parameters**:
    1. `fname`: File, filename, or generator to read.
    2. `dtype`: Data type of the resulting array; default is None, which means dtypes will be inferred.
    3. `delimiter`: String to separate values; default is ',' for CSV files.

- **Advantages**: Simplifies loading CSV files by setting default parameters, such as the delimiter to ',', and inferring data types.
- **Limitations**: Less flexible than directly using `genfromtxt` for complex CSV files that might not fit the common structure.

Reading other file types

1. **Sata Files:**
   Using SAS7BDAT:

```
from sas7bdat import SAS7BDAT

# Replace 'path/to/your/sas_file.sas7bdat' with the actual file path
with SAS7BDAT('path/to/your/sas_file.sas7bdat') as file:
    df = file.to_data_frame()
print(df.head())
```

Using Pandas:

```
import pandas as pd

# Replace 'path/to/your/sata_file.sata' with the actual file path
df = pd.read_sas('path/to/your/sata_file.sata')
print(df.head())
```

2. **Pickle Files**:
   Pickle files can be loaded in Python using the `pickle` module. This is useful for loading Python objects that were previously serialised using `pickle.dump()`. Here's how you can do it:

```
import pickle

# Replace 'path/to/your/file.pkl' with the actual file path
with open('path/to/your/file.pkl', 'rb') as file:
data = pickle.load(file)
print(data)
```

3. **HDF5**:
   Using h5py:

```
import h5py

# Replace 'path/to/your/file.h5' with the actual file path
with h5py.File('path/to/your/file.h5', 'r') as file:

# Assuming 'dataset_name' is the name of your dataset within the HDF5 file
data = file['dataset_name'][:]
print(data)
```

Using pandas:

```
import pandas as pd

# Replace 'path/to/your/file.h5' with the actual file path
# Replace 'your_key' with the key or path identifying the data in the HDF5
file
df = pd.read_hdf('path/to/your/file.h5', 'your_key')
print(df.head())
```

4. **MATLAB Files**:
   Using Scipy:

```
import scipy.io
# Load a .mat file
mat = scipy.io.loadmat('file_name.mat')
# Access data from the loaded .mat file
data = mat['variable_name']  # Replace 'variable_name' with the actual
variable name you want to access
```

SQL with Python

SQLAlchemy is a popular SQL toolkit and Object-Relational Mapping (ORM) library for Python. It provides a full suite of well-known enterprise-level persistence patterns, designed for efficient and high-performing database access. A connection can be established as follows:

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

Base = declarative_base()
```

```
# Connect to the database
engine = create_engine('sqlite:///mydatabase.db')
Base.metadata.create_all(engine)

# Create a session
Session = sessionmaker(bind=engine)
session = Session()
```

1. **Create:**

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    age = Column(Integer)

new_user = User(name='John Doe', age=30)
session.add(new_user)
session.commit()
```

2. **Read**:

```
# Fetch a single user by primary key
user = session.query(User).get(primary_key)

# Fetch all users
all_users = session.query(User).all()
for user in all_users:
    print(user.name, user.age)
```

3. **Update**:

```
# Fetch the user you want to update
user_to_update = session.query(User).get(primary_key)

# Update the fields
user_to_update.name = 'Jane Doe'
user_to_update.age = 32

# Commit the changes
session.commit()
```

5. **Delete**:

```
# Fetch the user you want to delete
user_to_delete = session.query(User).get(primary_key)
```

```
# Delete the user
session.delete(user_to_delete)

# Commit the changes
session.commit()
```

6. **Execute any SQL Query**:

```
# Import the necessary components from SQLAlchemy
from sqlalchemy import create_engine
from sqlalchemy.sql import text

# Define your database URL
database_url = 'your_database_url'

# Create an engine
engine = create_engine(database_url)

# Define your SQL query
sql_query = text("SELECT * FROM your_table_name")

# Execute the query directly using the engine
with engine.connect() as connection:
    result = connection.execute(sql_query)

# Fetch all results
rows = result.fetchall()

# Print all the columns
print(rows.keys())

# Iterate over the results
for row in rows:
        print(row)
```

7. **Execute SQL query in Pandas with sqlalchemy**:

```
# Import the necessary components from SQLAlchemy and pandas
from sqlalchemy import create_engine
import pandas as pd

# Define your database URL
database_url = 'your_database_url'

# Create an engine
engine = create_engine(database_url)
```

```
# Define your SQL query
        sql_query = "SELECT * FROM your_table_name"

        # Execute the query directly using pandas
        df = pd.read_sql_query(sql_query, engine)

        # Display the DataFrame
        print(df)
```

Cloud Computing

Cloud Vs On-premises

Feature	Cloud Computing	On-Premises Computing
Initial Cost	Low upfront cost	High upfront cost for infrastructure
Scalability	Highly scalable with demand	Limited by onsite hardware
Maintenance	Handled by service provider	Handled by organization's IT staff
Control	Less control over hardware and software	Full control over hardware and software
Security	High-level security managed by provider	Security is managed internally
Compliance	Dependent on provider	Easier to ensure compliance
Customization	Limited customization options	Highly customizable
Accessibility	Accessible from anywhere with internet	Access limited to on-site or VPN
Data Sovereignty	Dependent on provider's locations	Data stored on-premises
Upfront Investment	Minimal	Significant
Operational Expenses	Pay-as-you-go pricing model	Ongoing maintenance and upgrade costs

Compute Characteristics

1. **Scalability**: Ability to scale resources up or down based on demand.
2. **Performance**: High-performance computing capabilities for processing tasks.
3. **Virtualisation**: Supports virtual machines and containers for efficient resource utilisation.
4. **Cost-Effectiveness**: Pay-as-you-go pricing models to optimize costs.
5. **Availability**: High availability configurations to ensure continuous operation.

6. **Security**: Advanced security features to protect compute resources.
7. **Networking**: Integrated networking capabilities for connectivity and load balancing.
8. **Management**: Tools and services for managing and monitoring compute resources.
9. **Integration**: Easy integration with storage, database, and other cloud services.
10. **Elasticity**: Automatically adjusts computing capacity to meet workload demands.

Storage Characteristics

1. **Durability**: High durability to ensure data is not lost.
2. **Accessibility**: Data can be accessed anytime from anywhere.
3. **Scalability**: Ability to scale storage capacity up or down as needed.
4. **Security**: Robust security measures to protect data.
5. **Cost-Effectiveness**: Storage solutions offer various pricing models to optimize costs.
6. **Performance**: Fast access to stored data.
7. **Data Management**: Features for managing, archiving, and backing up data.
8. **Redundancy**: Multiple copies of data to prevent loss.
9. **Compliance**: Compliance with regulatory requirements.
10. **Integration**: Seamless integration with compute and database services.

Database Characteristics

1. **Scalability**: Databases can scale vertically and horizontally to handle load.
2. **Performance**: Optimized for fast query processing and data retrieval.
3. **Availability**: High availability configurations to minimize downtime.
4. **Security**: Strong security features to protect sensitive data.
5. **Management**: Tools for database management, monitoring, and optimisation.
6. **Data Integrity**: Ensures accuracy and consistency of data.
7. **Backup and Recovery**: Capabilities to backup data and recover from data loss.
8. **Replication**: Replication features for data distribution and redundancy.
9. **Data Modelling**: Supports various data models (relational, NoSQL, etc.).
10. **Transactions**: Support for ACID (Atomicity, Consistency, Isolation, Durability) properties for reliable transactions.

IaaS vs PaaS vs SaaS

IaaS (Infrastructure as a Service):

1. **Use Cases**: Ideal for companies needing complete control over their infrastructure, with the flexibility to configure and manage virtual machines, storage, and networking however they see fit. Common use cases include hosting complex applications, big data analysis, and disaster recovery.
2. **Characteristics**:

- Scalable resources on-demand.
- Complete control of the infrastructure.
- Pay-as-you-go pricing model.

3. **Advantages**:

- Reduces the need for physical hardware, lowering capital expenditure.
- High scalability and flexibility.
- Users maintain complete control over their infrastructure.

4. **Examples**: Amazon Web Services (AWS) EC2, Google Compute Engine (GCE), Microsoft Azure VMs.

PaaS (Platform as a Service):

1. **Use Cases**: Best suited for developers and development teams looking to streamline the development, testing, and deployment of applications. PaaS provides a platform with tools to develop, test, and host applications in the same environment.
2. **Characteristics**:
   - Development framework and tools provided.
   - Managed runtime environment.
   - Integrated development environment (IDE) support.
3. **Advantages**:
   - Simplifies the development process by providing a pre-configured platform.
   - Facilitates collaborative work even in geographically dispersed teams.
   - Reduces the amount of coding required.
4. **Examples**: Heroku, Google App Engine, Microsoft Azure App Services.

SaaS (Software as a Service):

1. **Use Cases**: Ideal for applications that require web or mobile access, such as email, customer relationship management (CRM) tools, and collaboration software. SaaS is used by businesses of all sizes for its convenience and ease of access.
2. **Characteristics**:
   - Software hosted on a remote server and accessed over the internet.
   - Subscription-based pricing model.
   - Managed by the service provider.
3. **Advantages**:
   - No installation, maintenance, or hardware required by the user.
   - Accessible from anywhere with an internet connection.
   - Scalable with flexible pricing models.
4. **Examples**: Google Workspace, Salesforce, Microsoft Office 365.

Private, Public, and Hybrid Cloud

1. **Private Cloud:**
   A Private Cloud is a cloud computing environment dedicated solely to one organization. It offers enhanced security and control, making it ideal for businesses with strict data privacy, compliance requirements, or unique configuration needs. Private clouds can be hosted on-premises or by a third-party service provider. The key advantage is the dedicated use of resources, allowing for customized security and performance.
2. **Public Cloud:**
   The Public Cloud is provided by third-party service providers and offers cloud services over the internet. Public clouds are shared infrastructures where resources, such as servers and storage, are owned and operated by the provider and shared among multiple tenants. Examples include Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). Public clouds are known for their scalability, reliability, and flexibility, allowing businesses to pay only for the resources they use.
3. **Hybrid Cloud:**
   A Hybrid Cloud combines private and public cloud infrastructures, allowing data and applications to be shared between them. This approach provides businesses with greater flexibility and more deployment options. Hybrid clouds are particularly beneficial for balancing between the need for scalability and the demand for regulatory compliance or data sovereignty. By leveraging a hybrid cloud, organisations can keep sensitive or critical workloads in the private cloud while utilizing the public cloud for scalable, less-sensitive tasks.

AWS Services Overview

1. **EC2 (Elastic Compute Cloud):**
   EC2 provides scalable computing capacity in the Amazon Web Services (AWS) cloud. It allows users to run virtual servers according to their needs. It eliminates the need to invest in hardware upfront, so you can develop and deploy applications faster.
2. **RedShift:**
   Amazon RedShift is a fully managed, petabyte-scale data warehouse service in the cloud. It allows you to run complex data analytics queries on large datasets and integrates well with popular BI tools. RedShift is designed for high performance analysis and reporting of data.
3. **S3 (Simple Storage Service):**
   Amazon S3 is an object storage service that offers industry-leading scalability, data availability, security, and performance. This means customers of all sizes and industries can use it to store and protect any amount of data for a range of use cases, such as data lakes, websites, mobile applications, backup and restore, archive, enterprise applications, IoT devices, and big data analytics.
4. **SageMaker**
   Amazon SageMaker is a fully managed service that provides every developer and data

scientist with the ability to build, train, and deploy machine learning (ML) models quickly. SageMaker removes the heavy lifting from each step of the machine learning process to make it easier to develop high-quality models.

5. **Kinesis:**
   Amazon Kinesis makes it easy to collect, process, and analyze real-time, streaming data. It allows you to process and analyze data as it arrives, which makes it possible to get timely insights and react quickly to new information.

Azure Service Overview

1. **Azure Blob Storage:** Azure Blob Storage is a scalable, object storage solution for the cloud. It is designed to store large amounts of unstructured data, such as text or binary data, making it ideal for serving images or documents directly to a browser, storing files for distributed access, streaming video and audio, and storing data for backup and restore, disaster recovery, and archiving.
2. **Azure VM (Virtual Machines):** Azure Virtual Machines (VMs) are one of several types of on-demand, scalable computing resources that Azure offers. Essentially, an Azure VM gives you the ability to deploy and manage virtual machines in the cloud with as much or as little control as you wish. You can use VMs for a wide variety of computing solutions including development and testing, running applications, and extending your data center.
3. **Azure SQL Database:** Azure SQL Database is a fully managed relational database service that offers SQL Server engine compatibility with built-in intelligence that learns app patterns and adapts to maximize performance, reliability, and data protection. It's a high-performance, durable, scalable, and secure database service that can handle large volumes of data and is accessible from anywhere in the world.
4. **Microsoft Service Fabric:** Microsoft Service Fabric is a distributed systems platform that makes it easy to package, deploy, and manage scalable and reliable microservices and containers. Service Fabric also addresses the significant challenges in developing and managing cloud applications. By using Service Fabric, developers can focus on implementing mission-critical, demanding workloads at scale using a microservices approach.
5. **Azure Stream Analytics:** Azure Stream Analytics is a real-time analytics and complex event-processing engine that is designed to analyze and process high volumes of fast streaming data from multiple sources simultaneously. It can be used to trigger alerts, feed information into a dashboard, start an action, or store data for later analysis. Stream Analytics is fully managed and can scale to meet demands.
6. **Azure Data Lake:** Azure Data Lake is a highly scalable and secure data lake for big data analytics. It combines the power of a Hadoop Distributed File System with integrated analytics tools. Data Lake makes it easy to store data of any size, shape, and speed, and do all types of processing and analytics across platforms and languages. It removes the complexities of ingesting and storing all of your data while making it faster to get up and running with batch, streaming, and interactive analytics.

7. **Azure Machine Learning:** Azure Machine Learning is a cloud-based environment you can use to train, deploy, automate, manage, and track ML models. It is designed to enable developers and data scientists to build and integrate machine learning models into applications without being experts in data science. Azure Machine Learning provides all the tools needed to rapidly build and deploy machine learning models, using high-level services and a scalable cloud infrastructure.

GCP Services Overview

1. **Google Cloud Storage:** Google Cloud Storage is a RESTful online file storage web service for storing and accessing data on Google Cloud Platform infrastructure. It is designed to make web-scale computing easier for developers. Key features include:
    - **Object storage** for companies of all sizes.
    - **Secure and durable storage** with automatic scalability.
    - **Data archiving, online backup, and disaster recovery**.
2. **Google Compute Engine:**
   Google Compute Engine offers scalable and flexible virtual machine computing capabilities in the cloud. With Compute Engine, you can:
    - **Create and run virtual machines** on Google's infrastructure.
    - **Choose from a wide range of machine types**.
    - **Pay only for what you use** with per-second billing.
3. **Google Cloud SQL:**
   Google Cloud SQL is a fully-managed database service that makes it easy to set up, maintain, manage, and administer your relational PostgreSQL, MySQL, and SQL Server databases in the cloud. Features include:
    - **Fully managed relational database** service.
    - **Automatic backups, replication, and failover** to ensure reliability and security.
    - **Scalability with minimal downtime**.
4. **Google BigQuery:**
   BigQuery is Google's fully managed, petabyte scale, low-cost analytics data warehouse. It is designed for:
    - **Speedy analysis of large datasets**.
    - **Integration with various data analysis tools and languages**.
    - **Serverless, so there is no infrastructure to manage**.
5. **Google DataFlow:**
   Google DataFlow is a fully managed streaming analytics service that minimises latency, processing time, and cost through autoscaling and batch processing. It is designed for:
    - **Real-time, complex event processing**.
    - **Integration with Apache Beam SDK** for developing and executing data processing pipelines.
    - **Simplifying the process of developing, managing, and executing a wide range of data processing patterns**.

6. **Google AutoML:**
   Google AutoML is a suite of Machine Learning products that enables developers with limited machine learning expertise to train high-quality models specific to their business needs. It offers:
   - **Custom ML models** with minimal effort and machine learning expertise.
   - **Various services** like AutoML Vision, AutoML Natural Language, and AutoML Tables.
   - **Integration with Google Cloud services** for seamless deployment and scalability.

Data Visualisation

Histogram

A histogram is a type of graph used in statistics and data visualization that represents the distribution of numerical data. It is a kind of bar chart that shows how many data points fall into specified ranges of values (bins). Each bin represents a range of values, and the height of each bar depicts the frequency or number of data points in that range.

Usage of Histograms:
Histograms are widely used in data analysis for various reasons:

1. **Understanding Distribution**: They help in understanding the underlying distribution of the data, whether it is normal, skewed, bimodal, etc.
2. **Identifying Outliers**: By visually inspecting the shape and spread of the histogram, one can identify any outliers or anomalies in the data.
3. **Comparing Datasets**: Histograms can be used to compare the distributions of two or more datasets, making it easier to spot differences in shape, center, and spread.
4. **Detecting Skewness**: They are useful in detecting the skewness of the data. A left-skewed distribution has a long left tail, while a right-skewed distribution has a long right tail.
5. **Estimating Density**: Histograms can be used to estimate the probability density function of the underlying distribution.

Key Components of a Histogram:

1. **Bins**: These are intervals that represent the x-axis. The range of values is divided into intervals, and each interval is a bin.
2. **Frequency**: The y-axis represents the frequency of observations within each bin. The height of the bar indicates how many observations fall into each bin.
3. **Width of Bins**: The width of each bin can be uniform or non-uniform. Choosing the right bin width is crucial for accurately representing the data.

Use Case:

1. **Number of Bins**: The choice of the number of bins can significantly affect the histogram's appearance and interpretability. Too few bins can oversimplify reality, while too many bins can complicate the understanding of the data.
2. **Bin Width**: Similarly, the width of the bins should be chosen carefully to balance detail and clarity.
3. **Outliers**: Be mindful of outliers as they can skew the histogram and give a misleading representation of the data distribution.

Histograms are a fundamental tool in data visualization, providing a quick and insightful overview of the distribution and characteristics of numerical data. They are essential for data analysis, helping in making informed decisions based on the data's distribution and trends.

Box plots

Box plots, also known as box-and-whisker plots, are a type of graph used in statistics and data visualization to display the distribution of a dataset. They are particularly useful for showing the central tendency, dispersion, and skewness of the data, as well as identifying outliers.

Components of a Box Plot:
A box plot consists of the following components:

- **Minimum**: The smallest value in the dataset, excluding outliers.
- **First Quartile (Q1)**: Also known as the lower quartile, it is the median of the lower half of the dataset.
- **Median (Q2)**: The middle value of the dataset.
- **Third Quartile (Q3)**: Also known as the upper quartile, it is the median of the upper half of the dataset.
- **Maximum**: The largest value in the dataset, excluding outliers.
- **Interquartile Range (IQR)**: The difference between the third and first quartiles (Q3 - Q1).
- **Whiskers**: Lines that extend from the first and third quartiles to the minimum and maximum values, respectively.
- **Outliers**: Data points that fall outside of the whiskers, often defined as 1.5 * IQR above the third quartile or below the first quartile.

Usage of Box Plots:
Box plots are used in various fields for exploratory data analysis because they provide a quick visual summary of the central tendency, variability, and shape of a distribution. Their specific uses include:

1. **Comparing Distributions**: Box plots are excellent for comparing the distributions of multiple datasets side by side.

2. **Identifying Outliers**: The clear representation of outliers helps in identifying unusual observations that may require further investigation.
3. **Summarising Data**: They provide a concise summary of the data, showing the range, median, and quartiles at a glance.
4. **Detecting Skewness**: The position of the median within the box, as well as the lengths of the whiskers, can indicate skewness in the data distribution.

Example:
Consider a dataset of test scores for two classes. A box plot for each class can quickly reveal differences in median scores, the spread of scores, and any outliers, thus providing valuable insights into the performance of each class.

Scatter Plot

A scatter plot is a type of data visualization that displays values for typically two variables for a set of data. The data is displayed as a collection of points, each having the value of one variable determining the position on the horizontal axis and the value of the other variable determining the position on the vertical axis.

Usage of Scatter Plots:
Scatter plots are used to observe and show relationships between two numeric variables. The basic uses of scatter plots include:

1. **Correlation Detection**: One of the primary uses of scatter plots is to observe and show the relationship between two variables. By visually inspecting a scatter plot, one can quickly determine if there is a positive correlation, negative correlation, or no correlation between the variables.
2. **Identifying Trends**: Scatter plots can help in identifying trends in data over time. If data points make a line or curve when plotted, this can indicate a trend.
3. **Outlier Detection**: Scatter plots make it easy to spot outliers in the data. An outlier is a data point that is significantly different from the other data points in the dataset.
4. **Data Distribution**: They can also give you a general idea about the distribution of the dataset, such as whether the data is tightly grouped, evenly spread, or clustered in parts.
5. **Comparing Groups**: If the data includes categories, scatter plots can be used to compare the groups. By using different colors or symbols for different groups, one can visually compare the groups.

Example of a Scatter Plot:
Imagine you have a dataset containing the heights and weights of a group of people. By plotting each person's weight on the X-axis and their height on the Y-axis, you create a scatter plot. This plot can help you understand if there's a relationship between height and weight within your dataset.

Line Plot

A **line plot** is a type of chart used in data visualization that displays information as a series of data points called 'markers' connected by straight line segments. It is one of the most basic and commonly used types of charts for showing trends over intervals or time series data. Line plots are particularly useful for visualizing the change in a variable over time or comparing multiple variables over the same time intervals.

Usage of Line Plots:
Line plots are widely used in various fields for different purposes, such as:

1. **Trend Analysis**: They are perfect for showing the trend of data over time, like stock market trends, temperature change over the years, etc.
2. **Comparing Changes**: Line plots can compare changes over the same period for more than one group. For example, comparing the sales of different products over the same time period.
3. **Forecasting**: In economics and finance, line plots are used to forecast future trends based on past data.
4. **Data Correlation**: They help in identifying the correlation between two variables over a period.

Key Components of a Line Plot:

1. **X-axis and Y-axis**: The horizontal axis (X-axis) typically represents the time interval or the independent variable, while the vertical axis (Y-axis) represents the magnitude of the variable being measured.
2. **Data Points/Markers**: These are specific values plotted on the chart.
3. **Line**: The line connecting the data points represents the trend or relationship between the variables.

Advantages of Line Plots:

1. **Simplicity**: They are straightforward to create and understand, making them accessible for a wide audience.
2. **Effectiveness in Showing Trends**: Line plots are highly effective in showing the direction and speed of a trend.
3. **Comparison**: They make it easy to compare multiple datasets on the same graph.

Use Case:

1. **Overplotting**: When dealing with large datasets, line plots can become cluttered and hard to read. It's essential to keep the chart simple and not overload it with too much information.
2. **Applicability**: Line plots are best suited for continuous data and trends. They might not be the best choice for categorical data or for comparing individual data points without a

clear trend or relationship.

HCL Palette

HCL (Hue-Chroma-Luminescence) is a color space that is particularly useful in data visualization for its ability to represent colors in a way that is more aligned with human perception than other color spaces like RGB or CMYK. Understanding each component of HCL can help in choosing the right color schemes for various data visualization tasks. Here's a breakdown of each component and their use cases in data visualization:

1. **Hue**:
    - **Definition**: Hue represents the type of color or the pure color. It is the aspect of colors that is described as "red", "blue", "green", etc.
    - **Use Cases**:
        1. **Categorical Data**: Different hues are used to distinguish categories within a dataset. For example, in a pie chart representing different market segments, each segment can be assigned a distinct hue.
        2. **Maps**: Hue variations can represent different regions, countries, or states in geographical data visualizations.
2. **Chroma:**
    - **Definition**: Chroma, sometimes referred to as saturation, measures the intensity or purity of a color. A color with high chroma is vivid, while a color with low chroma appears more muted.
    - **Use Cases**:
        1. **Highlighting Information**: High chroma colors can draw attention to key data points or outliers in scatter plots, line graphs, or bar charts.
        2. **Visual Emphasis**: Using varying chroma levels can create a visual hierarchy in your data, guiding the viewer's eye to the most important parts of the visualization.
3. **Luminescence:**
    - **Definition**: Luminescence refers to the brightness or lightness of a color. It determines how close a color is to white or black.
    - **Use Cases**:
        1. **Heatmaps**: Different levels of luminescence can represent varying degrees of a particular measure, such as temperature or density, with darker or lighter colors indicating higher or lower values, respectively.
        2. **Sequential Data Representation**: In charts where data needs to be represented in a sequence or gradient (e.g., from low to high), varying the luminescence can effectively show this progression.

Three Types of Color Scales in Data Visualisation

Data visualisation often employs color scales to represent or highlight data values. Understanding the types of color scales and their appropriate use cases is crucial for effective data representation. Here are the three primary types of color scales used in data visualisation:

1. **Qualitative Color Scales**

- **Description**: Qualitative color scales use distinct colors to represent different categories without implying any numerical order or value difference between categories. These scales are best for nominal or categorical data.
- **Use Case**: A bar chart showing different companies' market shares where each company is represented by a different color.

2. **Sequential Color Scales**

- **Description**: Sequential color scales use a range of colors that progress from light to dark (or vice versa) to represent an ordered sequence of values, usually from low to high. These scales are suitable for displaying data that has a natural order and where the magnitude of the data is important.
- **Use Case**: A heat map showing population density across different regions, with lighter colors indicating lower density and darker colors indicating higher density.

3. **Divergent Color Scales**

- **Description**: Divergent color scales use two contrasting color sequences to highlight values above or below a midpoint. They are ideal for visualizing data where the focus is on deviation from a median value.
- **Use Case**: A map showing temperature anomalies, with one color representing temperatures above the historical average and another color for temperatures below the average, converging at a neutral color for average temperatures.

Issues in Visualisation Plots

Visualizing data effectively is crucial for understanding trends, patterns, and outliers. However, certain practices can lead to confusion or misinterpretation of the data. Three common issues faced in data visualisation, especially when using dual axes, are:

1. **Dual Axes**: Dual axes plots use two y-axes, each corresponding to a different dataset. While this can be useful for comparing different scales, it can also lead to several issues:
    - **Scale Discrepancy**: If the scales are not carefully chosen, one dataset might appear more significant or volatile than it actually is, leading to misinterpretation.
    - **Confusion**: With two sets of data plotted against two y-axes, it can be challenging to discern which data corresponds to which axis, especially if the axes have similar ranges or if the plot is cluttered.

    - **Misleading Correlations**: When two datasets are plotted on the same graph but on different scales, it can falsely imply correlations or relationships between the datasets that do not exist.
2. **Axes of Evil**: This term humorously refers to misleading or poorly constructed axes. Common issues include:
    - **Truncated Axes**: Sometimes, axes are intentionally or unintentionally truncated to exaggerate trends or differences, misleading the viewer.
    - **Inconsistent Intervals**: Axes with irregular intervals can distort the perception of the data, making gradual changes appear sudden or vice versa.
    - **Lack of Context**: Axes without clear labels, units, or explanations can leave viewers guessing what the data represents, reducing the effectiveness of the visualisation.
3. **Sensory Overload**: Overloading a plot with too much information, colors, or text can overwhelm the viewer, leading to:
    - **Analysis Paralysis**: Too much information can make it difficult for viewers to focus on the key messages or trends in the data.
    - **Misinterpretation**: Excessive use of colors or overlapping data points can confuse viewers, leading to incorrect conclusions about the data.
    - **Aesthetic Over Function**: Sometimes, the desire to make a visually appealing plot can overshadow the need for clarity and accuracy, compromising the plot's integrity.

Cleaning Data

Constraints

1. **Data Type Constraints:**
   Data type constraints occur when there is data type mismatch. In pandas this is done using `astype`. This can be:
    1. **Type 1 to Type 2:** For example, `str` to `int`
    2. **Numerical or categorical:** Marriage status is categorical, age is numerical
2. **Unique Constraints:**
   Unique Constraints forces removal of duplicates. This can be done using `.drop_duplicated()`. It accepts `column_name` as a list of columns to be evaluated and `keep` being a parameter which accepts first, last and all which lists which value to be kept.
3. **Membership Constraints:**
   Categorise sometimes don't belong to the given list causing membership constraints such as a boolean category having 3rd label. `set` operation can be used to find anti-join which provides the categories causing the violations.
4. Categorical Value:
   Creating categories on a existing data. `pd.qcut(column, q=number_of_categories,`

`labels=category_names)` can be used for creating the categories. Misinterpreted categories can be changed using `replace()` .

5. **Data Range Constraints:**
   Dates ranging out the time period or values and ratings going above limit cause Data Range Constraints.
6. **Inconsistency**
7. **Completeness:**
   Missing data has to be handled for data completeness.
   `missingno.matrix(data_frame)` can be used to visualise the data. Missing values can be of three types:
   - **Missing Completely at Random (MCAR):** No systematic relationship between missing data and other values
   - **Missing At Random (MAR):** Systematic relationship between missing data and observed values
   - **Missing Not At Random (MNAR):** Systematic relationship between missing data and unobserved values