**Database Week Two Assignment**

**1.1**

To retrieve all data points (columns) from the "Expenses" table, you can use the following SQL query:

SQL

SELECT \* FROM Expenses;

The asterisk symbol (\*) is a wildcard character that represents all columns in the table. This query will return all columns and all rows from the "Expenses" table.

If you want to retrieve specific columns, you can replace the \* with the column names, separated by commas. For example:

SQL

SELECT ExpenseDate, Category, Amount FROM Expenses;

This query will return only the ExpenseDate, Category, and Amount columns from the "Expenses" table.

**1.2**

To select only specific columns relevant to your analysis, you can modify the query as follows:

SQL

SELECT Date, Category, Amount

FROM Expenses;

This query will only retrieve the Date, Category, and Amount columns from the "Expenses" table, which are relevant to analyzing spending patterns by category and date.

If you want to further filter the results, you can add a WHERE clause to the query. For example:

SQL

SELECT Date, Category, Amount

FROM Expenses

WHERE Category = 'Food' AND Date >= '2024-01-01';

This query will only retrieve the Date, Category, and Amount columns from the "Expenses" table where the Category is 'Food' and the Date is greater than or equal to '2024-01-01'. This can help you analyze your food expenses for the current year.

**1.3**

Here's a query to retrieve expenses charged between a specific date range:

SQL

SELECT \*

FROM Expenses

WHERE Date >= '2021-01-01' AND Date <= '2024-12-15';

In this query:

* Date is the column name representing the date of the expenses.
* '2021-01-01' and '2024-12-15' are the start and end dates of the range, respectively.
* >= and <= are used to include expenses on the start and end dates, respectively.

By using this query, you'll retrieve all expenses charged between January 1, 2021, and December 15, 2024.

Note: The date format used in the query is 'YYYY-MM-DD', which is the standard format for date literals in SQL.

**2.1**

SELECT \*

FROM Expenses

WHERE Category = 'Entertainment';

In this query:

* Category is the column name representing the category of the expenses.
* 'Entertainment' is the specific category we're interested in.

By using this query, you'll retrieve all expenses that have the category "Entertainment".

Note: The query is case-sensitive, so if the category values in your table are stored in a different case (e.g., "entertainment"), you'll need to adjust the query accordingly.

**2.2**

To find expenses with an amount greater than a certain value, you can use the following query:

SQL

SELECT \*

FROM Expenses

WHERE Amount > 50;

This query will retrieve all expenses with an amount greater than $50.

If you want to include expenses that are equal to $50, you can use the >= operator instead:

SQL

SELECT \*

FROM Expenses

WHERE Amount >= 50;

This query will retrieve all expenses with an amount greater than or equal to $50.

Note: The query assumes that the Amount column is of a numeric data type (such as INT or DECIMAL). If the Amount column is stored as a string, you may need to convert it to a numeric value in the query.

**2.3**

To refine your query to find expenses that meet multiple criteria, you can use the AND operator to combine conditions. Here's an example:

SQL

SELECT \*

FROM Expenses

WHERE Amount > 75 AND Category = 'Food';

In this query:

* Amount > 75 is the first condition, which selects expenses with an amount greater than $75.
* Category = 'Food' is the second condition, which selects expenses belonging to the "Food" category.
* AND is the operator that combines the two conditions, requiring both to be true for a row to be included in the results.

By using this query, you'll retrieve all expenses that have an amount greater than $75 and belong to the "Food" category.

You can also use the OR operator to combine conditions if you want to find expenses that meet either of the criteria. For example:

SQL

SELECT \*

FROM Expenses

WHERE Amount > 75 OR Category = 'Food';

This query will retrieve all expenses that have an amount greater than $75 or belong to the "Food" category, or both.

**2.4**

To find expenses belonging to one category or another, you can use the OR operator in your query. Here's an example:

SQL

SELECT \*

FROM Expenses

WHERE Category = 'Transportation' OR Category = 'Groceries';

This query will retrieve all expenses that belong to either the "Transportation" or "Groceries" category.

If you want to make the query more concise, you can use the IN operator instead:

SQL

SELECT \*

FROM Expenses

WHERE Category IN ('Transportation', 'Groceries');

Both queries will produce the same result, but the IN operator makes it easier to add more categories to the query if needed.

**2.5**

To display expenses unrelated to a specific category, you can use the NOT operator in your query. Here's an example:

SQL

SELECT \*

FROM Expenses

WHERE Category != 'Rent';

This query will retrieve all expenses that do not belong to the "Rent" category.

Alternatively, you can use the NOT IN operator to achieve the same result:

SQL

SELECT \*

FROM Expenses

WHERE Category NOT IN ('Rent');

Both queries will produce the same result, but the NOT IN operator makes it easier to exclude multiple categories if needed. For example:

SQL

SELECT \*

FROM Expenses

WHERE Category NOT IN ('Rent', 'Utilities', 'Insurance');

This query will retrieve all expenses that do not belong to the "Rent", "Utilities", or "Insurance" categorie

**3.1**

To display all expenses sorted by amount in a specific order, you can use the ORDER BY clause in your query. Here's an example:

SQL

SELECT \*

FROM Expenses

ORDER BY Amount DESC;

This query will retrieve all expenses sorted by amount in descending order (highest to lowest).

If you want to sort the expenses in ascending order (lowest to highest), you can use the ASC keyword instead:

SQL

SELECT \*

FROM Expenses

ORDER BY Amount ASC;

The ORDER BY clause can be used to sort the results by one or more columns. In this case, we're sorting by the Amount column, but you can add additional columns to the ORDER BY clause if needed.

For example, to sort by Category and then by Amount in descending order:

SQL

SELECT \*

FROM Expenses

ORDER BY Category, Amount DESC;

This query will first sort the expenses by Category, and then by Amount in descending order within each category.

**3.2**

To sort expenses based on multiple columns, you can list the columns in the ORDER BY clause, separated by commas. Here's an example:

SQL

SELECT \*

FROM Expenses

ORDER BY Date DESC, Category ASC;

This query will sort the expenses first by Date in descending order (newest to oldest), and then by Category in ascending order (alphabetical order).

By sorting by Date first, you'll see the most recent expenses at the top of the list. Then, within each date group, the expenses will be sorted by Category in alphabetical order.

You can add additional columns to the ORDER BY clause as needed, listing them in the order you want to apply the sorting.

For example, to sort by Date (descending), then by Category (ascending), and finally by Amount (descending):

SQL

SELECT \*

FROM Expenses

ORDER BY Date DESC, Category ASC, Amount DESC;

This query will show the most recent expenses (by date), grouped by category (in alphabetical order), and then sorted by amount (highest to lowest) within each category

**4.1**

Here's the SQL query to create the "Income" table:

SQL

CREATE TABLE Income (

income\_id INT PRIMARY KEY AUTO\_INCREMENT,

amount DECIMAL(10, 2) NOT NULL,

date DATE NOT NULL,

source VARCHAR(50) NOT NULL

);

This query creates a table named "Income" with the specified columns:

* income\_id: a primary key column with auto-incrementing integers
* amount: a decimal column to store the income amount with a maximum of 10 digits and 2 decimal places
* date: a date column to store the date of the income
* source: a varchar column to store the source of the income (e.g., salary, investment, etc.)

Note that the PRIMARY KEY constraint is used to define the income\_id column as the primary key, and the AUTO\_INCREMENT attribute is used to automatically generate a unique ID for each new record. The NOT NULL constraint is used to ensure that all columns have a value, preventing null or empty values from being inserted.

**4.2**

To add a new column named "category" of type VARCHAR(50) to the existing "Income" table, you can use the ALTER TABLE statement with the ADD COLUMN clause. Here's the SQL query:

SQL

ALTER TABLE Income

ADD COLUMN category VARCHAR(50);

This query will add a new column named "category" with a data type of VARCHAR(50) to the "Income" table. The "category" column will be able to store strings with a maximum length of 50 characters, allowing you to store income categories like "Salary", "Freelance Work", etc.

Note that this query will add the new column to the end of the existing table structure. If you want to add the column at a specific position, you can use the AFTER or BEFORE clause to specify the position. For example:

SQL

ALTER TABLE Income

ADD COLUMN category VARCHAR(50) AFTER source;

This query will add the "category" column after the "source" column in the table structure

**4.3**

To remove the "source" column from the "Income" table, you can use the ALTER TABLE statement with the DROP COLUMN clause. Here's the SQL query:

SQL

ALTER TABLE Income

DROP COLUMN source;

This query will permanently remove the "source" column from the "Income" table.

If you no longer need the entire "Income" table, you can use the DROP TABLE statement to permanently remove it from your database. Here's the SQL query:

SQL

DROP TABLE Income;

This query will permanently delete the entire "Income" table, including all its columns and data. Be careful when using the DROP TABLE statement, as it will irreversibly remove the table and its data.

Note that if you want to delete the data in the table but keep the table structure, you can use the TRUNCATE TABLE statement instead:

SQL

TRUNCATE TABLE Income;

This query will delete all the data in the "Income" table, but the table structure and columns will remain intact.