Aggregate Functions

Aggregate functions in SQL are functions that perform a calculation on a set of values and return a single value. These functions are commonly used with the **SELECT** statement to summarize data in a database. Here are some common aggregate functions in SQL:

1. **COUNT()**: Counts the number of rows in a specified column or the number of rows that meet a certain condition.

SELECT COUNT(column_name) FROM table_name;

1. **SUM()**: Calculates the sum of values in a numeric column.

SELECT SUM(column_name) FROM table_name;

1. **AVG()**: Calculates the average value of a numeric column.

SELECT AVG(column_name) FROM table_name;

1. **MIN()**: Retrieves the minimum value from a column.

SELECT MIN(column_name) FROM table_name;

1. **MAX()**: Retrieves the maximum value from a column.

SELECT MAX(column_name) FROM table_name;

1. **GROUP BY**: Groups rows that have the same values in specified columns into summary rows, often used with aggregate functions.

SELECT column1, COUNT(column2) FROM table_name GROUP BY column1;

1. **HAVING**: Specifies a search condition for a group or an aggregate function used with the **GROUP BY** clause.

SELECT column1, COUNT(column2) FROM table_name GROUP BY column1 HAVING COUNT(column2) > 1;

Subqueries

A subquery, also known as a nested query or inner query, is a query nested within another SQL query. Subqueries are enclosed in parentheses and are generally executed first, returning a result that is used in the main query.

Subqueries can be used in various parts of a SQL statement, such as the **SELECT**, **FROM**, **WHERE**, or **HAVING** clauses.

Subqueries are seen to be used most frequently with SELECT statement as shown below:

SELECT column_name

FROM table_name

WHERE column_name *expression operator*

(SELECT COLUMN_NAME from TABLE_NAME WHERE ...);

Views

Views in SQL are kind of virtual tables. A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database.

Sample Tables:

StudentDetails

S_ID	NAME	ADDRESS
1	Harsh	Kolkata
2	Ashish	Durgapur
3	Pratik	Delhi
4	Dhanraj	Bihar
5	Ram	Rajasthan

StudentMarks

ID	NAME	MARKS	AGE
1	Harsh	90	19
2	Suresh	50	20
3	Pratik	80	19
4	Dhanraj	95	21
5	Ram	85	18

CREATING VIEWS

We can create View using **CREATE VIEW** statement. A View can be created from a single table or multiple tables. **Syntax**:

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

view_name: Name for the View
table_name: Name of the table
condition: Condition to select rows

Examples:

Creating View from a single table:

- In this example we will create a View named DetailsView from the table StudentDetails. Query:

CREATE VIEW DetailsView AS
SELECT NAME, ADDRESS
FROM StudentDetails
WHERE S_ID < 5;

- To see the data in the View, we can query the view in the same manner as we query a table.

*SELECT * FROM DetailsView*;

Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

- In this example, we will create a view named StudentNames from the table StudentDetails. Query:

CREATE VIEW StudentNames AS

SELECT S_ID, NAME
FROM StudentDetails
ORDER BY NAME;

- If we now query the view as,

*SELECT * FROM StudentNames;*

Output:

S_ID	NAMES
2	Ashish
4	Dhanraj
1	Harsh
3	Pratik
5	Ram

- **Creating View from multiple tables**: In this example we will create a View named MarksView from two tables StudentDetails and StudentMarks. To create a View from multiple tables we can simply include multiple tables in the SELECT statement. Query:

CREATE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;

- To display data of View MarksView:

*SELECT * FROM MarksView;*

- Output:

NAME	ADDRESS	MARKS
Harsh	Kolkata	90
Pratik	Delhi	80
Dhanraj	Bihar	95
Ram	Rajasthan	85

LISTING ALL VIEWS IN A DATABASE

We can list View using the SHOW FULL TABLES statement or using the information_schema table. A View can be created from a single table or multiple tables.

Syntax (Using SHOW FULL TABLES):

use "database_name";
show full tables where table_type like "%VIEW";

Syntax (Using information_schema) :

*select * from information_schema.views where table_schema = "database_name";*

OR

select table_schema,table_name,view_definition from information_schema.views where table_schema = "database_name";

DELETING VIEWS

We have learned about creating a View, but what if a created View is not needed anymore? Obviously we will want to delete it. SQL allows us to delete an existing View. We can delete or drop a View using the DROP statement.

Syntax:

DROP VIEW view_name;

view_name: Name of the View which we want to delete.

For example, if we want to delete the View **MarksView**, we can do this as:

DROP VIEW MarksView;

UPDATING VIEWS

There are certain conditions needed to be satisfied to update a view. If any one of these conditions is **not** met, then we will not be allowed to update the view.

1. The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
2. The SELECT statement should not have the DISTINCT keyword.
3. The View should have all NOT NULL values.
4. The view should not be created using nested queries or complex queries.
5. The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

- We can use the **CREATE OR REPLACE VIEW** statement to add or remove fields from a view. **Syntax**:

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

- For example, if we want to update the view **MarksView** and add the field AGE to this View from **StudentMarks** Table, we can do this as:

CREATE OR REPLACE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS, StudentMarks.AGE
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;

Q1: What is an SQL trigger?

Answer:

An SQL trigger is a database object that is associated with a table and automatically executes a set of SQL statements when a specific event occurs on that table. Triggers are used to enforce business rules, maintain data integrity, and automate certain actions within a database.

They can be triggered by various events, such as inserting, updating, or deleting data in a table, and they allow you to perform additional operations based on those events.

create trigger [trigger_name]

[before | after]

{insert | update | delete}

on [table_name]

[for each row]

[trigger_body]

Different Trigger Types in SQL Server

Two categories of triggers exist:

1. DDL Trigger
2. DML Trigger
3. Logon Triggers

DDL Triggers

The Data Definition Language (DDL) command events such as Create_table, Create_view, drop_table, Drop_view, and Alter_table cause the DDL triggers to be activated.

DML Triggers

The Data uses manipulation Language (DML) command events that begin with Insert, Update, and Delete set off the DML triggers. corresponding to insert_table, update_view, and delete_table.

Logon Triggers

logon triggers are fires in response to a LOGON event. When a user session is created with a SQL Server instance after the authentication process of logging is finished but before establishing a user session, the LOGON event takes place. As a result, the PRINT statement messages and any errors generated by the trigger will all be visible in the SQL Server error log.

Suppose the Database Schema Query

mysql▷>desc Student;

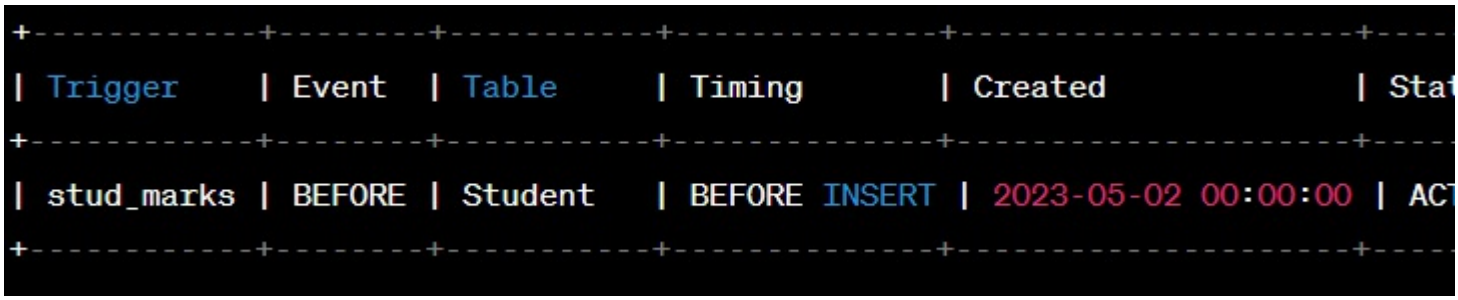
```
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| tid   | int(4)      | NO   | PRI | NULL    | auto_increment |
| name  | varchar(30) | YES  |     | NULL    |                |
| subj1 | int(2)      | YES  |     | NULL    |                |
| subj2 | int(2)      | YES  |     | NULL    |                |
| subj3 | int(2)      | YES  |     | NULL    |                |
| total | int(3)      | YES  |     | NULL    |                |
| per   | int(3)      | YES  |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
```

SQL Trigger to the problem statement.

```
CREATE TRIGGER stud_marks
BEFORE INSERT ON Student
FOR EACH ROW
SET NEW.total = NEW.subj1 + NEW.subj2 + NEW.subj3,
    NEW.per = (NEW.subj1 + NEW.subj2 + NEW.subj3) * 60 / 100;
```

Above SQL statement will create a trigger in the student database in which whenever subjects marks are entered, before inserting this data into the database, the trigger will compute those two values and insert them with the entered values. i.e.

Output



Q2: How do SQL triggers work?

Answer:

SQL triggers are defined using SQL statements and are associated with a specific table. When the defined trigger event (e.g., INSERT, UPDATE, DELETE) occurs on that table, the associated trigger code is executed automatically. The trigger code can consist of SQL statements that can manipulate data in the same or other tables, enforce constraints, or perform other actions.

Triggers are executed within the transaction scope, and they can be defined to execute either before or after the triggering event.

Q3: What are the benefits of using SQL triggers?

Answer:

The benefits of using SQL triggers include:

Data integrity: Triggers allow you to enforce complex business rules and constraints at the database level, ensuring that data remains consistent and accurate.

Automation: Triggers can automate repetitive or complex tasks by executing predefined actions whenever a specified event occurs. This reduces the need for manual intervention and improves efficiency.

Audit trails: Triggers can be used to track changes made to data, such as logging modifications in a separate audit table. This helps in auditing and maintaining a history of data changes.

Data validation: Triggers can perform additional validation checks on data before it is inserted, updated, or deleted, ensuring that only valid and conforming data is stored in the database.

Indexes

An index is a schema object. It is used by the server to speed up the retrieval of rows by using a pointer. It can reduce disk I/O(input/output) by using a rapid path access method to locate data quickly.

An index helps to speed up select queries and where clauses, but it slows down data input, with the update and the insert statements. Indexes can be created or dropped with no effect on the data. In this article, we will see how to create, delete, and use the INDEX in the database.

Creating an Index

Syntax

CREATE INDEX *index_name*

ON TABLE_name *column_name;*

where the **index** is the name given to that index **TABLE** is the name of the table on which that index is created and **column** is the name of that column for which it is applied.

For Multiple Columns

Syntax:

CREATE INDEX *index_name*

ON TABLE_name *(column1, column2,.....);*

For Unique Indexes

Unique indexes are used for the maintenance of the integrity of the data present in the table as well as for fast performance, it does not allow multiple values to enter into the table.

Syntax:

CREATE UNIQUE INDEX index

ON TABLE column;

When Should Indexes be Created?

- A column contains a wide range of values.
- A column does not contain a large number of null values.
- One or more columns are frequently used together in a where clause or a join condition.

When Should Indexes be Avoided?

- The table is small
- The columns are not often used as a condition in the query
- The column is updated frequently

Removing an Index

Remove an index from the data dictionary by using the **DROP INDEX** command.

Syntax

DROP INDEX index;

To drop an index, you must be the owner of the index or have the **DROP ANY INDEX** privilege.

Altering an Index

To modify an existing table's index by rebuilding, or reorganizing the index.

ALTER INDEX IndexName

ON TableName REBUILD;

Confirming Indexes

You can check the different indexes present in a particular table given by the user or the server itself and their uniqueness.

Syntax:

*SELECT * from USER_INDEXES;*

It will show you all the indexes present in the server, in which you can locate your own tables too.

Renaming an Index

You can use the system-stored procedure sp_rename to rename any index in the database.

Syntax:

EXEC sp_rename

index_name,

new_index_name,

N'INDEX';

SQL Server Database

Syntax:

DROP INDEX TableName.IndexName;

DataTypes in SQL

DATATYPE	DESCRIPTION	U
CHAR	string(0-255), can store characters of fixed length	C
VARCHAR	string(0-255), can store characters up to given length	V
BLOB	string(0-65535), can store binary large object	B
INT	integer(-2,147,483,648 to 2,147,483,647)	IN
TINYINT	integer(-128 to 127)	TI
BIGINT	integer(-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)	B
BIT	can store x-bit values. x can range from 1 to 64	B
FLOAT	Decimal number - with precision to 23 digits	FI
DOUBLE	Decimal number - with 24 to 53 digits	D
BOOLEAN	Boolean values 0 or 1	B
DATE	date in format of YYYY-MM-DD ranging from 1000-01-01 to 9999-12-31	D
YEAR	year in 4 digits format ranging from 1901 to 2155	Y

DataTypes can be of two types i.e Signed and Unsigned . Signed are those where –ve and +ve both values is possible . Unsigned are those where there is no such conflict i.e age , salary.

e.g. let's consider TINYINT (-128 to 127) , if we write TINYINT unsigned then it will be of size (0 to 255) i.e no need for –ve considerations.

Q) What are SQL commands ? Types of them.

SQL (Structured Query Language) is a domain-specific language used for managing and manipulating relational databases. SQL commands are instructions used to interact with a database.

1. **Data Query Language (DQL):**
   - DQL commands are used to retrieve data from one or more tables in a database.
   - Example: **SELECT** - retrieves data from one or more tables.
2. **Data Definition Language (DDL):**
   - DDL commands are used to define, modify, and delete database objects such as tables and indexes.
   - Examples: **CREATE**, **ALTER**, **DROP**.
3. **Data Manipulation Language (DML):**
   - DML commands are used to manipulate data stored in the database.
   - Examples: **INSERT**, **UPDATE**, **DELETE**.

Data Control Language (DCL):

-
  - DCL commands are used to control access to data within the database.
  - Examples: **GRANT**, **REVOKE**.

1. **Transaction Control Language (TCL):**
   - TCL commands are used to manage transactions within a database.
   - Examples: **COMMIT**, **ROLLBACK**, **SAVEPOINT**.

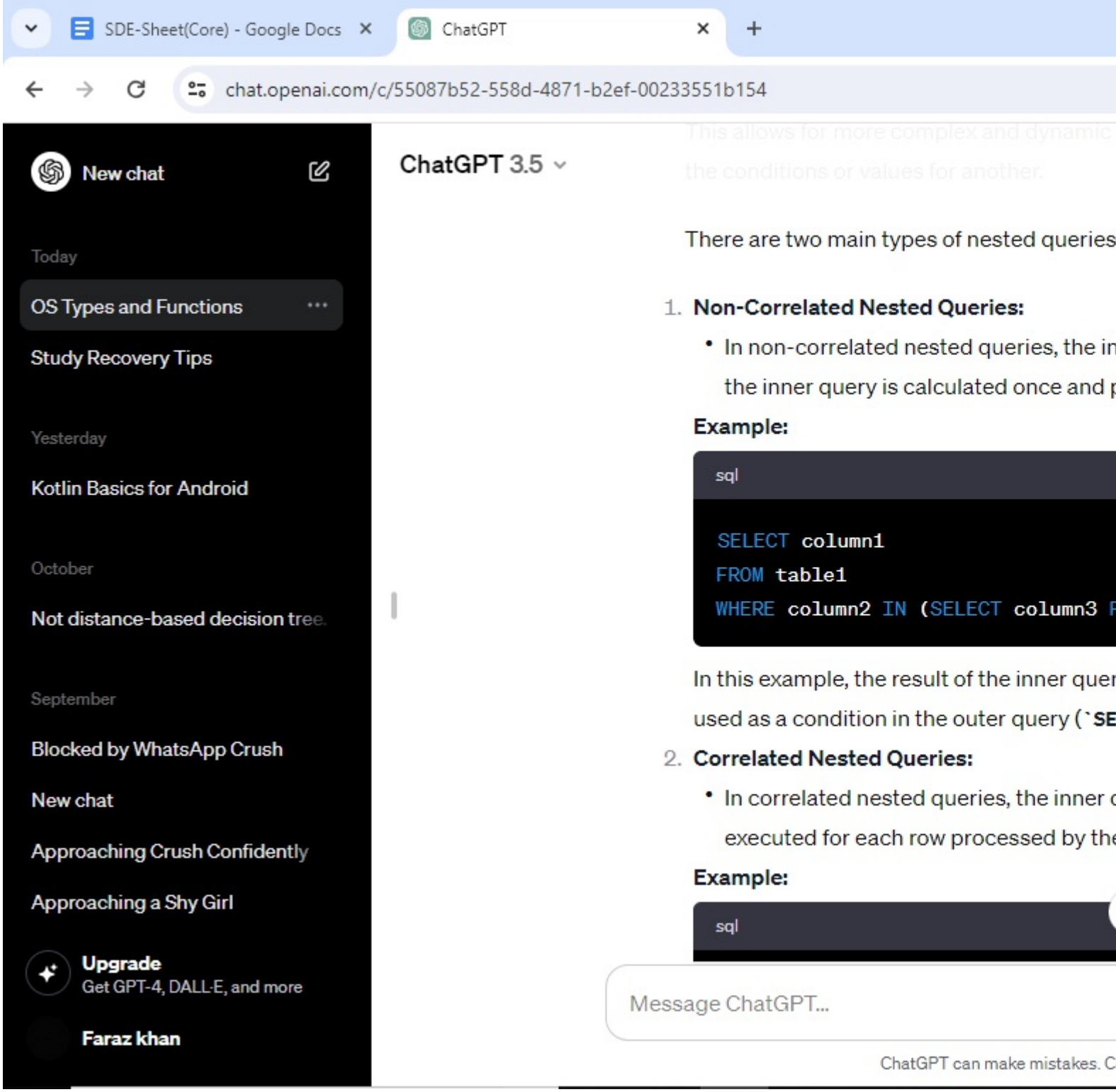
Q) Nested Queries in SQL ?

Nested queries in SQL involve placing one SQL query (known as the inner query) inside another SQL query (known as the outer query). The result of the inner query is used as a condition in the outer query. This allows for more complex and dynamic queries, where the result of one query is used to determine the conditions or values for another.

There are two main types of nested queries: correlated and non-correlated.

1. **Non-Correlated Nested Queries:**
   - In non-correlated nested queries, the inner query is independent of the outer query. The result of the inner query is calculated once and passed to the outer query.

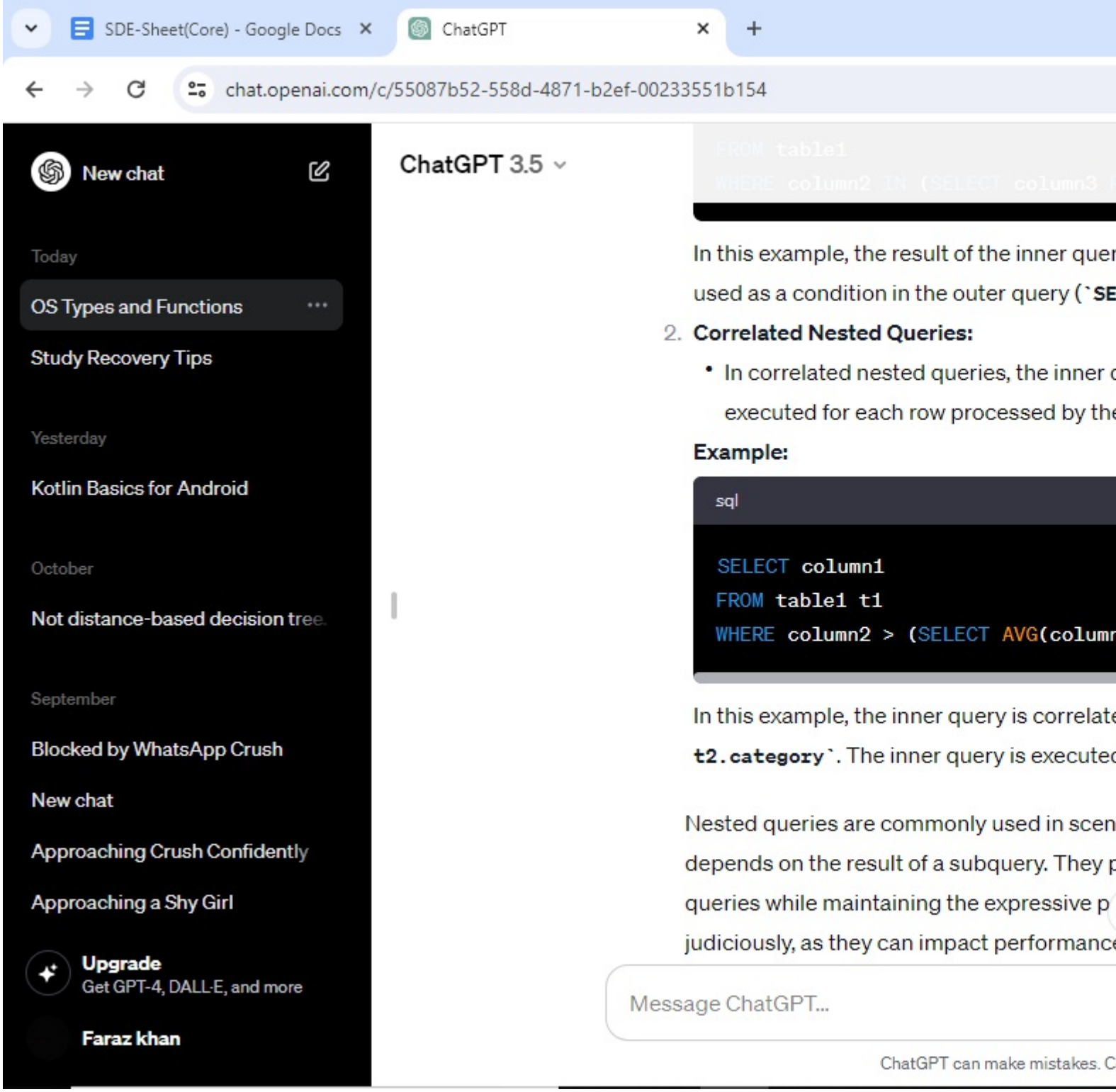
Example:



In this example, the result of the inner query (**SELECT column3 FROM table2 WHERE condition**) is used as a condition in the outer query (**SELECT column1 FROM table1 WHERE column2 IN ...**).

1. **Correlated Nested Queries:**
    - In correlated nested queries, the inner query depends on the outer query. The inner query is executed for each row processed by the outer query.

Example:



In this example, the inner query is correlated to the outer query through the condition **t1.category = t2.category**. The inner query is executed for each row of the outer query.

Q) What is JOIN ? Explain types of JOINs.

In Database Management Systems (DBMS), a JOIN operation combines rows from two or more tables based on a related column between them. JOINs are essential for retrieving data that spans multiple tables in a relational database. There are several types of JOINs, each serving a specific purpose. Here are the main types of JOINs:

1. **INNER JOIN:**
   - The INNER JOIN keyword selects records that have matching values in both tables.
   - Only the rows with common values in the specified columns from both tables are included in the result set.

SELECT *

FROM table1

INNER JOIN table2 ON table1.column_name = table2.column_name;

1. **LEFT (OUTER) JOIN:**
    - The LEFT JOIN keyword returns all records from the left table (table1), and the matched records from the right table (table2). If there is no match, NULL values are returned for columns from the right table.

```
SELECT *

FROM table1

LEFT JOIN table2 ON table1.column_name = table2.column_name;
```

1. **RIGHT (OUTER) JOIN:**
    - The RIGHT JOIN keyword returns all records from the right table (table2), and the matched records from the left table (table1). If there is no match, NULL values are returned for columns from the left table.

```
SELECT *

FROM table1

RIGHT JOIN table2 ON table1.column_name = table2.column_name;
```

1. **FULL (OUTER) JOIN:**
    - The FULL JOIN keyword returns all records when there is a match in either the left (table1) or the right (table2) table records. If there is no match, NULL values are returned for columns from the table without a match.

```
SELECT *

FROM table1

FULL JOIN table2 ON table1.column_name = table2.column_name;
```

1. **CROSS JOIN:**
    - The CROSS JOIN keyword returns the Cartesian product of the two tables, i.e., all possible combinations of rows from both tables.
    - It does not require a specific condition to join tables.

```
SELECT *

FROM table1

CROSS JOIN table2;
```

1. **SELF JOIN:**
    - A self join is a regular join, but the table is joined with itself. It is used to combine rows within the same table based on related columns.

```
SELECT *

FROM table1 t1

INNER JOIN table1 t2 ON t1.column_name = t2.column_name;
```