SQL Views and its Operations:

In SQL, a view is a virtual table based on the result-set of an SQL SELECT statement. It does not store data itself but instead provides a dynamic window into the data stored in one or more underlying base tables.

Key characteristics of SQL views:

Virtual Table:

Views appear and can be queried like regular tables, but they do not physically store data.

Defined by a Query:

A view's content is derived from a SELECT query that defines which columns and rows from the underlying tables should be included.

Dynamic Data:

When a view is accessed, the underlying SELECT query is executed, and the data is retrieved dynamically from the base tables. Any changes to the base tables are reflected in the view. A view is created with the CREATE VIEW statement.

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

CREATE VIEW Brazil Customers AS SELECT CustomerName, ContactName FROM Customers WHERE Country = 'Brazil';

SELECT * FROM Brazil Customers;

we will create a View named DetailsView from the table StudentDetails.

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

SELECT * FROM DetailsView;

we will create a view named StudentNames from the table StudentDetails.

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

create a view named StudentNames from the table StudentDetails.

CREATE VIEW StudentNames AS SELECT S_ID, NAME FROM StudentDetails ORDER BY NAME;

Creating a View From Multiple Tables:

we will create a View MarksView that combines data from bothtables StudentDetails and StudentMarks. To create a View from multiple tables we can simply include multiple tables in the SELECT statement.

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

CREATE VIEW Products Above Average Price AS SELECT ProductName, Price FROM Products WHERE Price > (SELECT AVG(Price) FROM Products);

SQL CREATE OR REPLACE VIEW Syntax:

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

Inserting Data into Views

INSERT INTO DetailsView(NAME, ADDRESS) VALUES ("Suresh", "Gurgaon");

SQL DROP VIEW Syntax:

DROP VIEW view_name;

DROP VIEW [Brazil Customers];

SQL WITH clause also known as Common Table Expressions (CTEs) comes in to make multiple nested subqueries, aggregations, and joins easier.

With clause example:

Example 1: Finding Employees with Above-Average Salary

EmployeeID	Name	Salary
100011	Smith	50000
100022	Bill	94000
100027	Sam	70550
100845	Walden	80000
115585	Erik	60000
1100070	Kate	69000

```
WITH temporaryTable (averageValue) AS (
    SELECT AVG(Salary)
    FROM Employee
)
        SELECT EmployeeID,Name, Salary
        FROM Employee, temporaryTable
        WHERE Employee.Salary > temporaryTable.averageValue;
```

Output

EmployeeID	Name	Salary
100022	Bill	94000
100845	Walden	80000

Recursion examples in SQL:

Example 1: Count Up Until Three

The first example we'll explore is count until three.

```
WITH countUp AS (SELECT 1 as n
                 UNION ALL
                 SELECT n+1 FROM countUp WHERE n < 3)
SELECT * FROM countUp
```

Running a recursive with statement to execute count until three command.

The base query returns number 1, the recursive query takes it under the countUp name and produces number 2, which is the input for the next recursive

call. When the recursive query returns an empty table n >= 3, the results from the calls are stacked together.

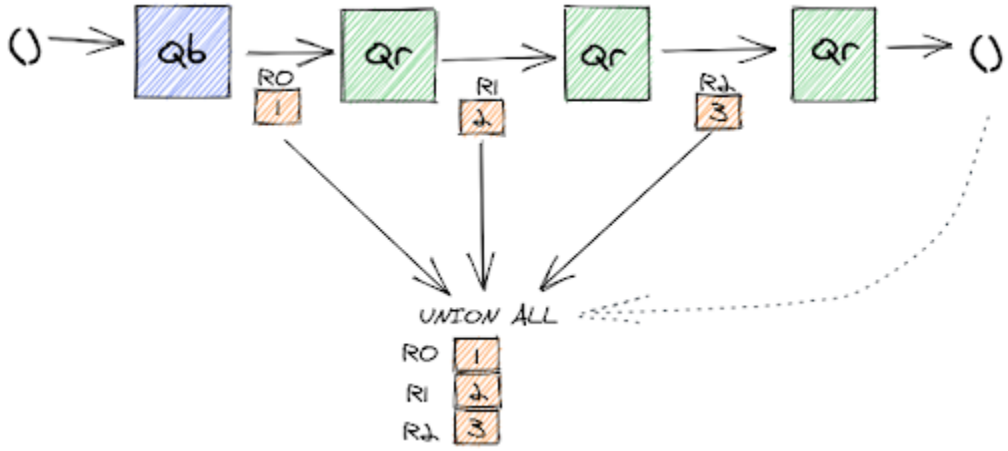
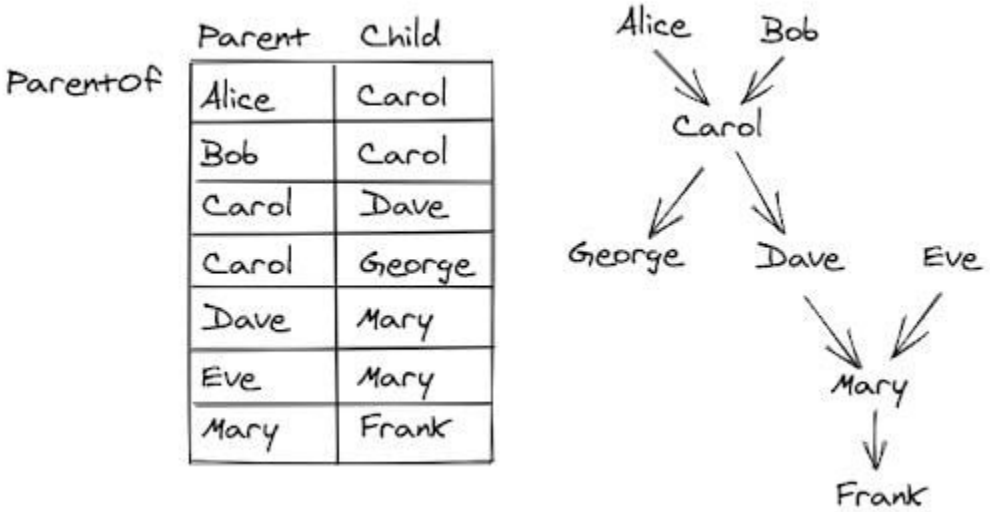


Illustration of the results from the call stacked together.

Counting up like that, however, can only go that far. There is a limit for recursion. It defaults to 100, but it could be extended with the MAXRECURSION option. In practice, however, it could be a bad idea to crank the recursion limit up. Graphs might have cycles and limited recursion depth can be a good defense mechanism to prevent a poorly behaving query: OPTION (MAXRECURSION 200).

Example 2: Finding Ancestors

Let's look at another example, finding a person's ancestors.



Using recursion to find the ancestors of a person.

```
WITH Ancestor AS (SELECT parent AS p FROM ParentOf WHERE child = 'Frank'
                  UNION ALL
                  SELECT parent FROM Ancestor, ParentOf
                  WHERE Ancestor.p = ParentOf.child)
SELECT * FROM Ancestor
```

Recursive common table expression to find the ancestors of a person.

The base query finds Frank's parent, Mary, and then the recursive query takes this result under the Ancestor name and finds Mary's parents, who are Dave and Eve. This process continues until we can't find any more parents.

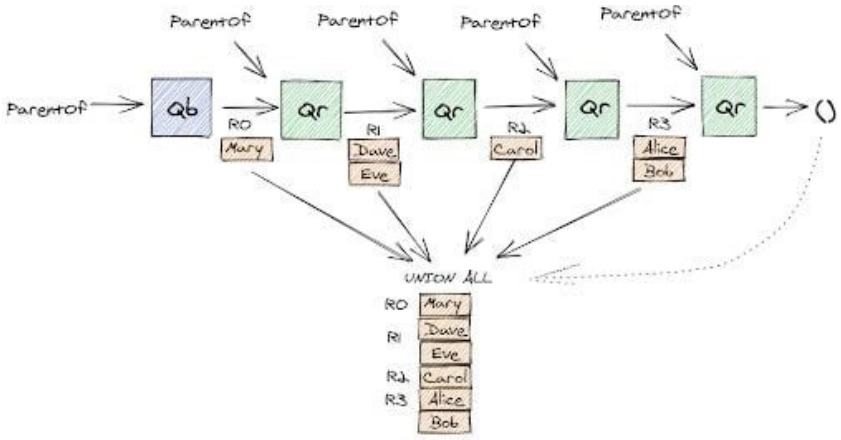


Illustration of the results from the recursion to find the ancestors of a person.

ParentOf

Parent	Child	BirthYear
Alice	Carol	1945
Bob	Carol	1945
Carol	Dave	1970
Carol	George	1972
Dave	Mary	2000
Eve	Mary	2000
Mary	Frank	2020

A data table that includes the birth year to find the parents of a person.

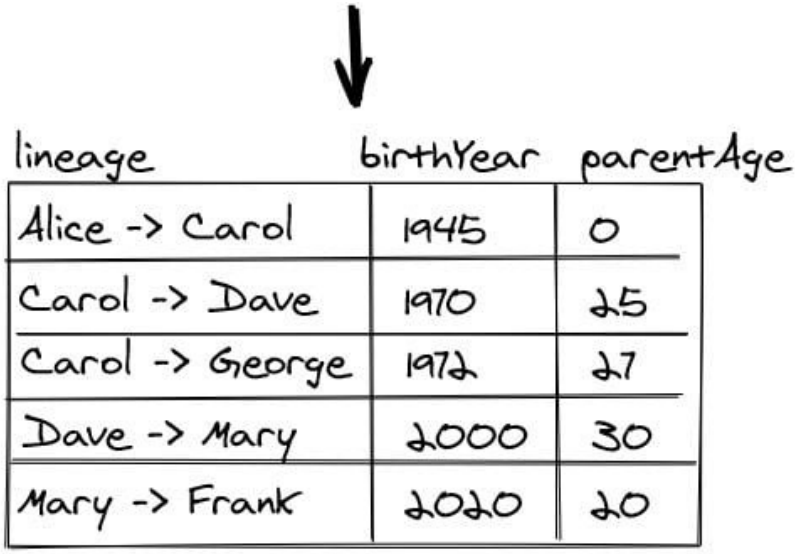
This tree traversal query could be the basis from which you augment the query with some other information of interest. For example, having a birth year in the table, we can calculate how old the parent was when the child was born. Our next query can do exactly that, along with showing lineages. To do that, it traverses the tree from top to bottom. parentAge is zero in the first row because we don't know when Alice was born from the data we have.

```
WITH Descendant
AS (SELECT parent + ' -> ' + child AS lineage, child AS c, birthYear, 0 AS parentAge
    FROM ParentOf WHERE parent = 'Alice'

    UNION ALL

    SELECT parent + ' -> ' + child AS lineage, child, ParentOf.birthYear,
           ParentOf.birthYear - Descendant.birthYear
    FROM Descendant, ParentOf
    WHERE Descendant.c = ParentOf.parent)
SELECT lineage, birthYear, parentAge FROM Descendant
```

Running a recursion to find the birth year of a person and their ancestors.



lineage	birthYear	parentAge
Alice -> Carol	1945	0
Carol -> Dave	1970	25
Carol -> George	1972	27
Dave -> Mary	2000	30
Mary -> Frank	2020	20

Table representing the results from the recursion to find the birth year and ancestors of a person.

The takeaway is that the recursive query references the result of our base query or a previous invocation of the recursive query. The chain stops when the recursive query returns an empty table.

****