Rekursive Abfragen mithilfe von allgemeinen Tabellenausdrücken (CTE)

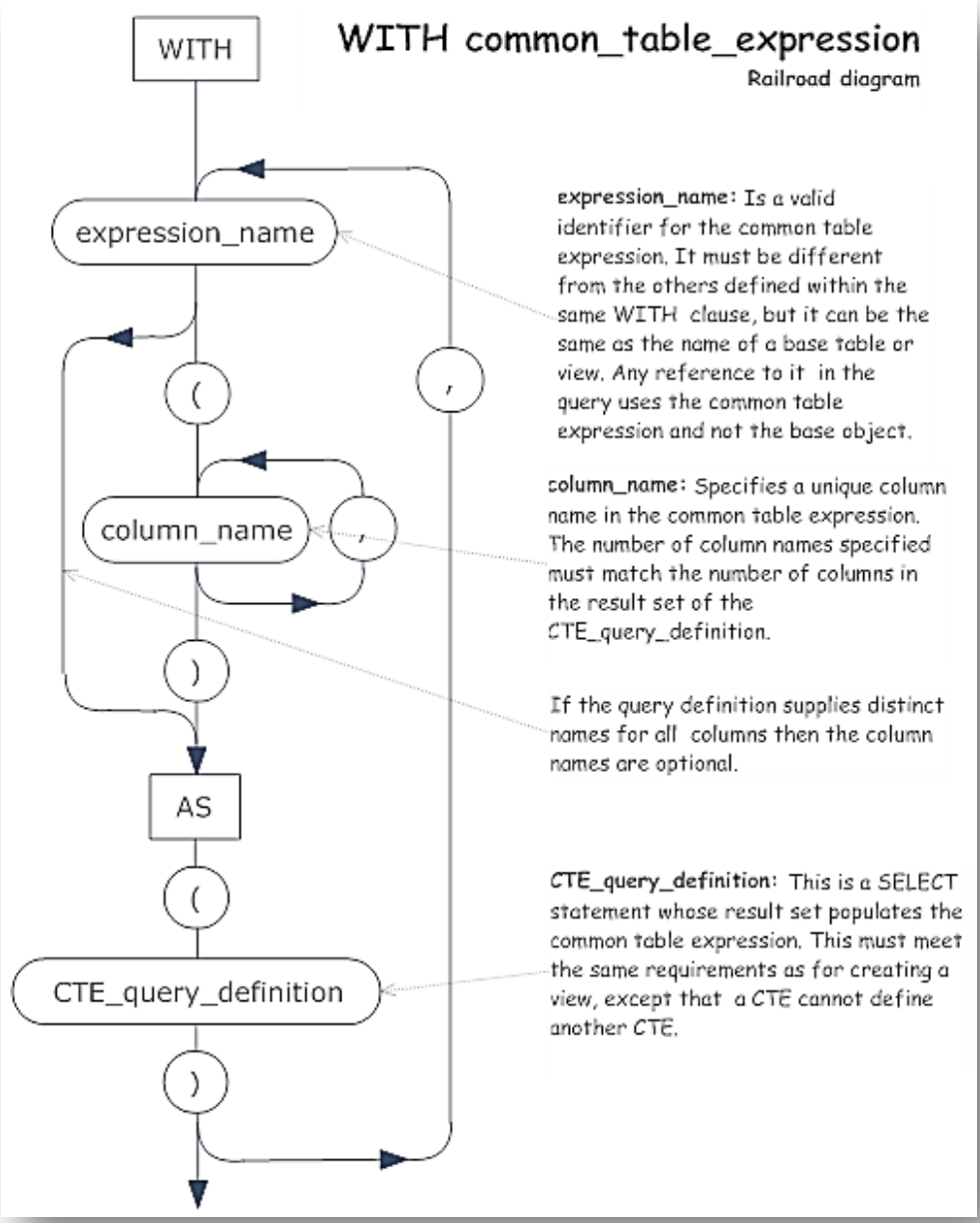
```
WITH RECURSIVE cte_count (n)
AS (
  SELECT 1
  UNION ALL
  SELECT n + 1
  FROM cte_count
  WHERE n < 3
)
SELECT n
FROM cte_count;
```

Anchor member

Recursive member

Termination condition

Query that uses CTE



Ein rekursiver CTE ist ein allgemeiner Tabellenausdruck, der sich selbst referenziert. Ihr lateinischer Wurzel, *recurrere*, bedeutet "zurücklaufen".

Eine rekursive Routine besteht aus drei Hauptteilen:

- Aufruf - Dies ist der Teil Ihrer Abfrage oder Ihres Programms, der die rekursive Routine aufruft.
- Rekursive Aufruf der Routine - Dies ist der Teil der Routine, die sich selbst aufruft.
- Beendigungsüberprüfung - Dies ist die Logik, die sicherstellt, dass wir schliesslich aufhören, die Routine aufzurufen und Antworten zu geben.

Ein allgemeiner Tabellenausdruck (CTE, ***Common Table Expression***) hat den Vorteil, dass er auf sich selbst verweisen kann, wodurch ein rekursiver CTE erstellt wird. Ein rekursiver CTE ist ein Ausdruck, bei dem ein ursprünglicher CTE wiederholt ausgeführt wird, um so lange Teilmengen von Daten zurückzugeben, bis das vollständige Resultset abgerufen wurde.

Ab SQL Server 2005 wird eine Abfrage als rekursive Abfrage bezeichnet, wenn sie auf einen rekursiven CTE verweisen. Die Rückgabe hierarchischer Daten ist eine häufige Verwendung rekursiver Abfragen, z. B. Anzeigen von Mitarbeitern in einem Organisationsdiagramm oder von Daten in einem Stücklistenszenario, in dem ein übergeordnetes Produkt eine oder mehrere Komponenten aufweist und diese Komponenten wiederum möglicherweise Unterkomponenten enthalten oder Komponenten anderer übergeordneter Elemente sind.
Ein rekursiver CTE kann erheblich zur Vereinfachung des Codes beitragen, der zur Ausführung einer rekursiven Abfrage innerhalb einer SELECT-, INSERT-, UPDATE-, DELETE- oder CREATE VIEW-Anweisung benötigt wird. In früheren Versionen von SQL Server setzt eine rekursive Abfrage üblicherweise die Verwendung temporärer Tabellen, Cursorn und Logik voraus, um den Ablauf der rekursiven Schritte zu steuern.

Die Struktur des rekursiven CTE in Transact-SQL entspricht der von rekursiven Routinen in anderen Programmiersprachen. Obwohl eine rekursive Routine in anderen Sprachen einen Skalarwert zurückgibt, kann ein rekursiver CTE mehrere Zeilen zurückgeben.

Ein rekursiver CTE besteht aus drei Elementen:

1. **Aufruf der Routine**. Der erste Aufruf des rekursiven CTE besteht aus einem oder mehreren *CTE_query_definitions*, die durch die Operatoren UNION ALL, UNION, EXCEPT oder INTERSECT miteinander verknüpft sind. Da diese Abfragedefinitionen das Basisresultset der CTE-Struktur bilden, werden sie als Ankerelemente bezeichnet. *CTE_query_definitions* werden als Ankerelemente betrachtet, sofern sie nicht auf das CTE selbst verweisen. Alle Ankerelement-Abfragedefinitionen müssen vor der ersten rekursiven Elementdefinition positioniert werden, und ein UNION ALL-Operator muss verwendet werden, um das letzte Ankerelement mit dem ersten rekursiven Element zu verknüpfen.

2. **Rekursiver Aufruf der Routine**. Der rekursive Aufruf schliesst eine oder mehrere *CTE_query_definitions* ein, die durch UNION ALL-Operatoren verknüpft sind, welche auf das CTE selbst verweisen. Diese Abfragedefinitionen werden als rekursive Elemente bezeichnet.

3. **Beendigungsprüfung.** Die Beendigungsprüfung ist implizit: Die Rekursion wird angehalten, wenn aus dem vorherigen Aufruf keine Zeilen mehr zurückgegeben werden.

Hinweis:

Ein fehlerhaft zusammengestellter rekursiver CTE kann zu einer Endlosschleife führen. So wird z. B. eine Endlosschleife erzeugt, wenn die Abfragedefinition des rekursiven Elements dieselben Werte sowohl für die übergeordneten als auch für die untergeordneten zurückgibt. Beim Testen der Ergebnisse einer rekursiven Abfrage können Sie die Anzahl der für eine bestimmte Anweisung zulässigen Rekursionsstufen einschränken, indem Sie den MAXRECURSION-Hinweis und einen Wert zwischen 0 und 32767 in der OPTION-Klausel der INSERT-, UPDATE-, DELETE- oder SELECT-Anweisung verwenden.

Pseudocode und Semantik

Die Struktur des rekursiven CTE muss mindestens ein Ankerelement und ein rekursives Element enthalten. Der folgende Pseudocode zeigt die Komponenten eines einfachen rekursiven CTE, der ein Ankerelement und ein rekursives Element enthält.

```
WITH cte_name ( column_name ,...n )
AS
(
       CTE_query_definition      -- Anchor member is defined, only called once

       UNION ALL

       CTE_query_definition      -- Recursive member is defined referencing cte_name.
)
SELECT * FROM cte_name           -- Statement using the CTE, Innvocation
```

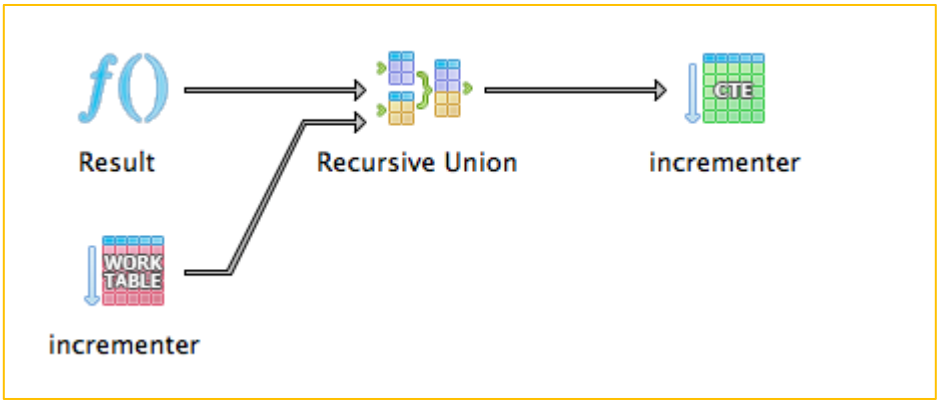
Die Semantik der rekursiven Ausführung ist folgendermassen:

1. Aufteilen des CTE-Ausdrucks in Ankerelemente und rekursive Elemente.
2. Ausführen der Ankerelemente zum Erzeugen des ersten Aufrufs oder Basisresultsets (T_0).
3. Ausführen der rekursiven Elemente mit T_i als Eingabe und T_{i+1} als Ausgabe.
4. Wiederholen von Schritt 3, bis ein leeres Set zurückgegeben wird.
5. Rückgabe des Resultsets. Das ist ein UNION ALL von T_0 bis T_n.

Hier ein rekursiver CTE, der von 1 bis 50 zählt:

```
WITH cte AS
       (SELECT 1 AS n  -- anchor member, only called once, T0

         UNION ALL

        SELECT n+1   -- recursive member, Tn
          FROM cte
        WHERE n<50   -- terminator
       )
SELECT n FROM cte OPTION (MAXRECURSION 200);     -- Statement using the CTE, Innvocation
```

Unter de
r Haube baut die Datenbank eine Tabelle auf, die nach diesem rekursiven CTE unter Verwendung von Vereinigungen benannt ist:



Beispiel:
Das folgende Beispiel zeigt die Semantik der rekursiven CTE-Struktur durch Rückgabe einer hierarchischen Mitarbeiterliste, beginnend mit dem hochrangigsten Mitarbeiter im Unternehmen. Die Anweisung zur Ausführung des CTE schränkt das Resultset auf Mitarbeiter *‚Bob'* ein. Erklärungen hinsichtlich der Codeausführung begleiten das Beispiel.

```
CREATE TABLE Person(
       id int PRIMARY KEY NOT NULL,
       Nachname varchar(15) NULL,
       idVorgesetzter int NULL,
       FOREIGN KEY (idVorgesetzter) REFERENCES Person (id)
);
INSERT INTO Person (id, Nachname, idVorgesetzter) VALUES (1, 'Don', NULL);
INSERT INTO Person (id, Nachname, idVorgesetzter) VALUES (2, 'Bob', NULL);
INSERT INTO Person (id, Nachname, idVorgesetzter) VALUES (3, 'Niels', 1);
INSERT INTO Person (id, Nachname, idVorgesetzter) VALUES (4, 'Keith', 2);
INSERT INTO Person (id, Nachname, idVorgesetzter) VALUES (5, 'John', 4);
INSERT INTO Person (id, Nachname, idVorgesetzter) VALUES (6, 'Steve', 1);
INSERT INTO Person (id, Nachname, idVorgesetzter) VALUES (7, 'Tim', 2);
INSERT INTO Person (id, Nachname, idVorgesetzter) VALUES (8, 'Peter', 4);
INSERT INTO Person (id, Nachname, idVorgesetzter) VALUES (9, 'Maria', 8);
INSERT INTO Person (id, Nachname, idVorgesetzter) VALUES (10, 'Jaclyn', 3);
INSERT INTO Person (id, Nachname, idVorgesetzter) VALUES (11, 'Anita', 3);
```

Die Daten in der Tabelle ‚Person':

Ergebnisse | Meldungen

	id	name	idVorgesetzter
1	1	Don	NULL
2	2	Bob	NULL
3	3	Niels	1
4	4	Keith	2
5	5	John	4
6	6	Steve	1
7	7	Tim	2
8	8	Peter	4
9	9	Maria	8
10	10	Jaclyn	3
11	11	Anita	3

Microsoft

Oracle

Die CTE: (Es sollen alle Unterstellten von Bob ermittelt werden!)

```
with PersonH(id, Nachname, idVorgesetzter, Level) as (

-- this is the anchor member
select id, Nachname, idVorgesetzter, null as Level
  from Person
    where Nachname='Bob'

union all

-- this is the recursive member
select e.id, e.Nachname, e.idVorgesetzter, Level
  from Person e
```

Ergebnisse | Meldungen

	id	name	idVorgesetzter
1	2	Bob	NULL

Ergebnisse | Meldungen

	id	name	idVorgesetzter
1	1	Don	NULL
2	2	Bob	NULL
3	3	Niels	1
4	4	Keith	2
5	5	John	4
6	6	Steve	1
7	7	Tim	2
8	8	Peter	4
9	9	Maria	8
10	10	Jaclyn	3
11	11	Anita	3

```
join PersonH eh on e.idVorgesetzter=eh.id
)
-- Statement that executes the CTE
select id, Nachname, idVorgesetzter, Level from PersonH OPTION (MAXRECURSION 200)
```

Das ganze CTE-SQL ergibt jetzt:

Ergebnisse | Meldungen

	id	name	idVorgesetzter
1	2	Bob	NULL
2	4	Keith	2
3	7	Tim	2
4	5	John	4
5	8	Peter	4
6	9	Maria	8

Erklärungen zum Beispiel-CTE:

1. Der rekursive CTE ***PersonH*** definiert ein Ankerelement und ein rekursives Element.

2. Das Ankerelement gibt das Basisresultset T_0 zurück. Dies ist der hochrangigste Mitarbeiter (Bob) im Geschäftsbereich, d. h. ein Mitarbeiter, der nicht gegenüber einem Vorgesetzten rechenschaftspflichtig ist.

   Hier ist das vom Ankerelement zurückgegebene Resultset:

   Ergebnisse | Meldungen

	id	name	idVorgesetzter
1	2	Bob	NULL

3. Das rekursive Element gibt alle Mitarbeiter aus dem Resultset des Ankerelements zurück. Dies wird durch eine Verknüpfungsoperation zwischen der *Person-Tabelle* und dem CTE *PersonH* erreicht. Es ist dieser Verweis auf den CTE selbst, der den rekursiven Aufruf verursacht. Basierend auf dem Mitarbeiter im CTE *PersonH* als Eingabe (T_i) gibt die Verknüpfung (join PersonH eh on e.idVorgesetzter = eh.id) als Ausgabe (T_{i+1}) zurück, also die Mitarbeiter, für die (T_i) der Vorgesetzter ist. Deshalb gibt die erste Iteration des rekursiven Elements dieses Resultset zurück:

   Ergebnisse | Meldungen

	id	name	idVorgesetzter
1	4	Keith	2
2	7	Tim	2

4. Das rekursive Element wird wiederholt aktiviert. Die zweite Iteration des rekursiven Elements verwendet das zweizeilige Resultset von Schritt 3 als Eingangswert und gibt dieses Resultset zurück:

   Ergebnisse | Meldungen

	id	name	idVorgesetzter
1	5	John	4
2	8	Peter	4

   Die dritte Iteration des rekursiven Elements verwendet das einzeilige Resultset von oben als Eingangswert und gibt dieses Resultset zurück:

   Ergebnisse | Meldungen

	id	name	idVorgesetzter
1	9	Maria	8

   Die vierte Iteration des rekursiven Elements verwendet das vorherige Zeilenset. Dieser Prozess wird wiederholt, bis das rekursive Element ein leeres Resultset zurückgibt.

5. Das durch Ausführen der Abfrage zurückgegebene endgültige Resultset ist das Ergebnis der UNION-Operation aller Resultsets, die von den Anker- und rekursiven Elementen erzeugt wurden.
   Hier das vom Beispiel zurückgegebene vollständige Resultset:

Noch ein Beispiel

```
with Dates as (
   select cast('2021-01-01' as date) as CalendarDate
   union all
   select dateadd(day , 1, CalendarDate) AS CalendarDate
   from Dates
   where dateadd (day, 1, CalendarDate)<= dateadd(year, 1, getdate())
)
select
   CalendarDate,
   CalendarYear=year(CalendarDate),
   CalendarQuarter=datename(quarter, CalendarDate),
   CalendarMonth=month(CalendarDate),
   CalendarWeek=datepart(wk, CalendarDate),
   CalendarDay=day(CalendarDate),
   CalendarMonthName=datename(month, CalendarDate),
   CalendarDayOfYear=datename(dayofyear, CalendarDate),
   Weekday=datename(weekday, CalendarDate),
   DayOfWeek=datepart(weekday, CalendarDate)
from Dates OPTION (MAXRECURSION 1500)
```

Results | Messages

	CalendarDate	CalendarYear	CalendarQuarter	CalendarMonth	CalendarWeek	CalendarDay	CalendarMonthName	CalendarDayOfYear	Weekday	DayOfWeek
1	2021-01-01	2021	1	1	1	1	Januar	1	Freitag	5
2	2021-01-02	2021	1	1	1	2	Januar	2	Samstag	6
3	2021-01-03	2021	1	1	1	3	Januar	3	Sonntag	7
4	2021-01-04	2021	1	1	2	4	Januar	4	Montag	1
5	2021-01-05	2021	1	1	2	5	Januar	5	Dienstag	2
6	2021-01-06	2021	1	1	2	6	Januar	6	Mittwoch	3
7	2021-01-07	2021	1	1	2	7	Januar	7	Donnerstag	4
8	2021-01-08	2021	1	1	2	8	Januar	8	Freitag	5
9	2021-01-09	2021	1	1	2	9	Januar	9	Samstag	6
10	2021-01-10	2021	1	1	2	10	Januar	10	Sonntag	7
11	2021-01-11	2021	1	1	3	11	Januar	11	Montag	1
12	2021-01-12	2021	1	1	3	12	Januar	12	Dienstag	2
13	2021-01-13	2021	1	1	3	13	Januar	13	Mittwoch	3
14	2021-01-14	2021	1	1	3	14	Januar	14	Donnerstag	4
15	2021-01-15	2021	1	1	3	15	Januar	15	Freitag	5

Noch ein Beispiel

```
WITH CTEMonth AS
(
       SELECT 1 AS MonNum
   UNION ALL
       SELECT MonNum + 1 -- add month number to 1 recursively
       FROM CTEMonth WHERE MonNum < 12 -- just to restrict the monthnumber upto 12
)
SELECT
      MonNum,
      DATENAME(MONTH, DATEADD(MONTH,MonNum,0)- 1)[MonthName] -- function to list the monthname.
FROM CTEMonth
```

Ergebnisse | Meldungen

	MonNum	MonthName
1	1	Januar
2	2	Februar
3	3	März
4	4	April
5	5	Mai
6	6	Juni
7	7	Juli
8	8	August
9	9	September
10	10	Oktober
11	11	November
12	12	Dezember

Noch ein Beispiel

```
USE tempdb;
GO
DROP TABLE dbo.Test;
GO
CREATE TABLE dbo.Test
  (data INTEGER NOT NULL,);
GO
CREATE CLUSTERED INDEX c ON dbo.Test (data);
GO
INSERT dbo.Test WITH (TABLOCK)
        (data)
SELECT TOP 50000000 ROW_NUMBER() OVER (ORDER BY (SELECT 0))/117329
FROM master.sys.columns C1, master.sys.columns C2, master.sys.columns C3, master.sys.columns C4;
GO
SET STATISTICS TIME ON;
```

SQL-Server neu starten!!

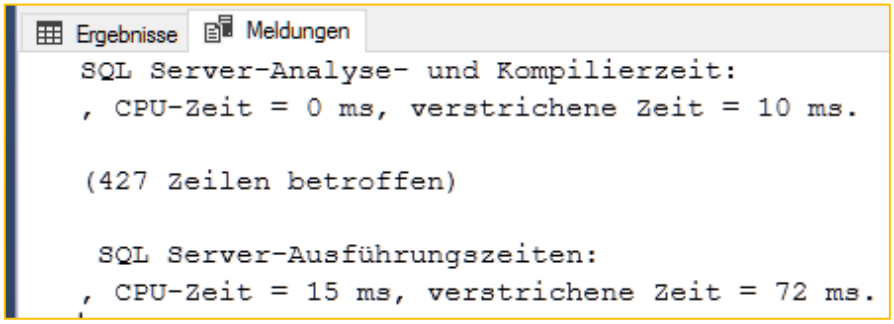
```
SELECT DISTINCT [data] FROM dbo.Test;
```

Ergebnisse | Meldungen

```
   SQL Server-Analyse- und Kompilierzeit:
   , CPU-Zeit = 184 ms, verstrichene Zeit = 184 ms.

   (427 Zeilen betroffen)

    SQL Server-Ausführungszeiten:
   , CPU-Zeit = 6966 ms, verstrichene Zeit = 1027 ms.
```

SQL-Server neu starten!!

```
WITH RecursiveCTE
AS (
    SELECT [data] = MIN(T.data) FROM dbo.Test T
    UNION   ALL
    SELECT R.data
    FROM (
            SELECT T.data,
                    rn = ROW_NUMBER() OVER (ORDER BY T.data)
            FROM dbo.Test T
            JOIN RecursiveCTE R
                    ON  R.data < T.data
          ) R
          WHERE R.rn = 1
    )
SELECT * FROM RecursiveCTE OPTION (MAXRECURSION 0);

SET STATISTICS TIME OFF;

DROP TABLE dbo.Test;
```



The recursive CTE is 14.3 times more efficient

Noch ein Beispiel

In high school, most of us first become aware of elementary number theory by being introduced to the Fibonacci sequence.
To refresh your memory, this is the sequence of numbers that progresses as follows:

1, 1, 2, 3, 5, 8, 13, 21, …

Start with the numbers 1 and 1, and the next number in the sequence is calculated as the sum of the prior two numbers. The recursion formula for this is (FN=Fibonacci Number):

$FN_n = FN_{n-1} + FN_{n-2}$ for n>2

```
DECLARE @fn1 DECIMAL(38,0);
DECLARE @fn2 DECIMAL(38,0);
SELECT @fn1 = 1, @fn2 = 1;

WITH FiboSequence AS (
    -- Put both initial numbers into the anchor leg
    SELECT
        n     = 1,
        fn    = @fn1,
        [n-1] = CAST(NULL AS BIGINT),
        Ratio = CAST(NULL AS FLOAT)

    UNION ALL

    SELECT
        n     = 2,
        fn    = @fn2,
        [n-1] = @fn1,
        -- Multiply * 1. to avoid integer division
        Ratio = 1. * @fn2/@fn1
```

```
    UNION ALL
    -- Recursive leg
    SELECT
        n     = n + 1,
        fn    = fn + [n-1],
        [n-1] = fn,
       Ratio  = (fn + [n-1])/CAST(fn AS FLOAT)
    FROM FiboSequence WHERE n BETWEEN 2 AND 180
)
SELECT n, fn, Ratio FROM FiboSequence OPTION (MAXRECURSION 180)
```

Fibonacci numbers grow quite large very quickly, hence the reason for CASTing them to the BIGINT datatype. Even so, you can see that we must stop prior to reaching the default recursion limit of SQL.

```
n    fn                                  Ratio
1    1                                   NULL
2    1                                   1
3    2                                   1
4    3                                   2
5    5                                   1.5
6    8                                   1.66666666666667
7    13                                  1.6
8    21                                  1.625
9    34                                  1.61538461538462
<snip>
90   2880067194370816120                 1.61803398874989
91   4660046610375530309                 1.61803398874989
92   7540113804746346429                 1.61803398874989
```

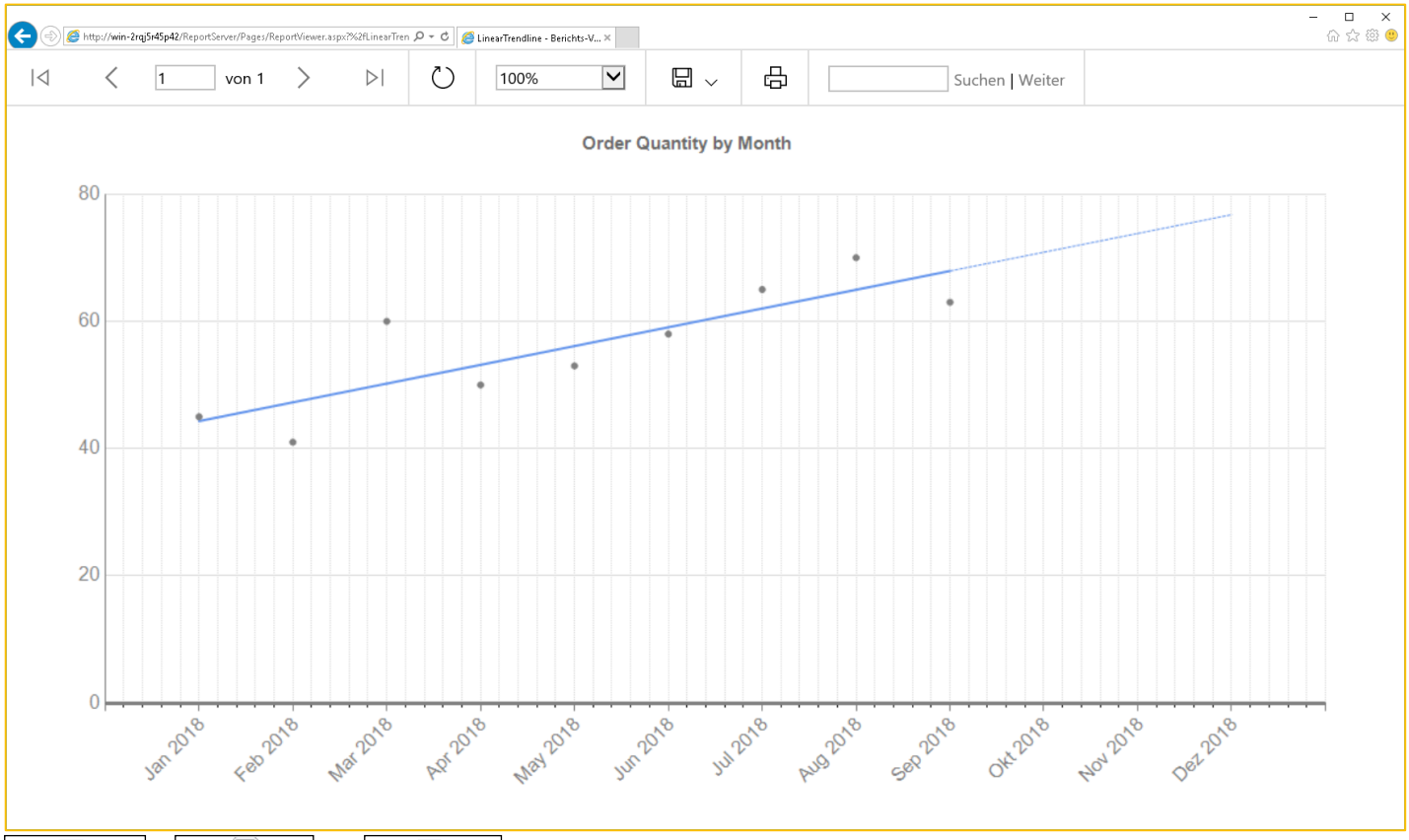
The significance of the ratio column is to show how the ratio of successive Fibonacci numbers converges on the Golden Ratio (1.6180339887). So how's that for a history lesson?

Einfacher:

```
WITH fibonacci (n, fib_n, next_fib_n) AS (
          SELECT 1, 0, 1
          UNION ALL
          SELECT n+1, next_fib_n, fib_n + next_fib_n
          FROM fibonacci
          WHERE n<20 )

SELECT * FROM fibonacci;
```

```
WITH factorial(n, fact) AS (
          SELECT 0, 1
          UNION ALL
          SELECT n + 1, fact * (n+1)
          FROM factorial
          WHERE n<10 )

SELECT * from factorial;
```

Noch ein Beispiel (FH-Niveau): Vorhersagen (lineare Regression)



LinearTrendline.rdl Trendlinie.sql dbTrendline.bak

```
SELECT ID = 1, OrderMonth = 'Jan 2018', OrderQuantity = 45, Trend = CONVERT(DECIMAL(38, 10),NULL)
INTO Regression
UNION ALL
SELECT ID = 2, OrderMonth = 'Feb 2018', OrderQuantity = 41, Trend = CONVERT(DECIMAL(38, 10),NULL)
UNION ALL
SELECT ID = 3, OrderMonth = 'Mar 2018', OrderQuantity = 60, Trend = CONVERT(DECIMAL(38, 10),NULL)
UNION ALL
SELECT ID = 4, OrderMonth = 'Apr 2018', OrderQuantity = 50, Trend = CONVERT(DECIMAL(38, 10),NULL)
UNION ALL
SELECT ID = 5, OrderMonth = 'May 2018', OrderQuantity = 53, Trend = CONVERT(DECIMAL(38, 10),NULL)
UNION ALL
SELECT ID = 6, OrderMonth = 'Jun 2018', OrderQuantity = 58, Trend = CONVERT(DECIMAL(38, 10),NULL)
UNION ALL
SELECT ID = 7, OrderMonth = 'Jul 2018', OrderQuantity = 65, Trend = CONVERT(DECIMAL(38, 10),NULL)
UNION ALL
SELECT ID = 8, OrderMonth = 'Aug 2018', OrderQuantity = 70, Trend = CONVERT(DECIMAL(38, 10),NULL)
UNION ALL
SELECT ID = 9, OrderMonth = 'Sep 2018', OrderQuantity = 63, Trend = CONVERT(DECIMAL(38, 10),NULL);
GO

DECLARE @sample_size INT;
DECLARE @intercept DECIMAL(38, 10);
DECLARE @slope DECIMAL(38, 10);
DECLARE @sumX  DECIMAL(38, 10);
DECLARE @sumY  DECIMAL(38, 10);
DECLARE @sumXX DECIMAL(38, 10);
DECLARE @sumYY DECIMAL(38, 10);
DECLARE @sumXY DECIMAL(38, 10);

SELECT
  @sample_size = COUNT(*)
 ,@sumX = SUM(ID)
 ,@sumY = SUM([OrderQuantity])
 ,@sumXX = SUM(ID*ID)
 ,@sumYY = SUM([OrderQuantity]*[OrderQuantity])
 ,@sumXY = SUM(ID*[OrderQuantity])
FROM Regression;
```

```
--SELECT SampleSize= @sample_size,SumRID=@sumX,SumOrderQty=@sumY,SumXX=@sumXX,SumYY=@sumYY,SumXY=@sumXY;
-- Berechnen der Steigung und des Schnittpunktes
SET @slope = CASE WHEN @sample_size = 1
    THEN 0 -- Vermeiden Sie Division durch Null Fehlerr
    ELSE (@sample_size * @sumXY - @sumX * @sumY) / (@sample_size * @sumXX - POWER(@sumX,2))
    END;
SET @intercept = (@sumY - (@slope*@sumX)) / @sample_size;

-- Trendlinie berechnen
UPDATE Regression
SET Trend = (@slope*ID) + @intercept;

--SELECT * FROM Regression;

/* Berechne vorhergesagte Werte */
-- Erstellen Sie eine vollständige Liste von 12 Monaten
WITH CTE_AllIDs AS
(
 SELECT TOP 12 ID = ROW_NUMBER() OVER (ORDER BY (SELECT NULL))
 FROM sys.columns
)
SELECT
  c.ID
 ,OrderMonth = CASE WHEN r.ID IS NOT NULL
                 THEN r.OrderMonth
                   -- Funktion, um den kurzen Monatsnamen und das Jahr zu erhalten
                 ELSE CONCAT(LEFT(DATENAME(MONTH, CONVERT(DATE, CONCAT('2018','-',c.ID,'-','01'))),3),' 2018')
               END
 ,OrderQuantity
 ,Trend
 ,Forecast = CASE WHEN Trend IS NOT NULL AND c.ID <> (SELECT MAX(ID) FROM Regression)
        THEN NULL
       -- Für den letzten tatsächlichen Wert (September in diesem Beispiel) möchten wir, dass die Prognose den gleichen Wert wie
die Trendlinie hat (anstelle von NULL). Dies verhindert eine Lücke in den Liniendiagrammen in SSRS.
       WHEN Trend IS NOT NULL AND c.ID = (SELECT MAX(ID) FROM Regression)
        THEN Trend
       -- Wenn kein Trend gefunden wird, können wir eine Prognose berechnen.
           -- Wir müssen jedoch auch prüfen, ob der Monat, für den wir die Prognose berechnen, nach dem tatsächlichen Werten
liegt.
           -- Angenommen, wir haben keine Werte für Januar, dann wollen wir auch keine Prognose für Januar berechnen. Nur für
die letzten 3 Monate des Jahres in diesem Beispiel.
       WHEN Trend IS NULL AND c.ID > (SELECT MAX(ID) FROM Regression)
        THEN (@slope * (c.ID % 100)) + @intercept
       ELSE NULL
       END
FROM  CTE_AllIDs   c
LEFT JOIN Regression r ON c.ID = r.ID;
```

Links

https://docs.microsoft.com/en-us/sql/t-sql/queries/with-common-table-expression-transact-sql

https://dwaincsql.com/2014/03/23/common-table-expressions-in-sql/ sehr gut

TSQL Tips and Tricks (mssqltips.com) sehr gut

Aufgabe 0

Erstellen sie die Tabelle *Person* wie oben definiert und testen sie das beschriebene CTE!
Erfassen sie weitere 2 Hierarchiestufen in der Tabelle *Person*!

Aufgabe 1

Erstellen sie eine Dokumentverwaltung welche wie oben (Person) eine *Hierachie der Dokumente* aufisten kann. Das heisst ein Dokument kann auf ein anderes Dokument verweisen.
Erfassen sie mindestens 3 Hierarchiestufen in der Tabelle *Dokument*!

Also in etwa so:

```
CREATE Dokument(
       id int IDENTITY(1,1) NOT NULL,
       Dokuname varchar(15) NULL,
       LinkZuDoku int NULL,
       . . . . . ,
       . . . . . ,
       . . . . . ,
)
```

Ihre CTE-Abfrage lautet:

Aufgabe 2

Erstellen sie ein CTE der die ungeraden Zahlen bis 1001 auflistet.

Tipp:

```
WITH cte AS
       (SELECT 1 AS n -- anchor member, only called once

         UNION ALL

        SELECT n+1    -- recursive member
          FROM cte
        WHERE n<50    -- terminator
       )
SELECT n FROM cte (MAXRECURSION 200);     -- Statement using the CTE, Innvocation
```

Aufgabe 3

Erstellen sie ein CTE das doppelte Einträge in einer Tabelle löscht.

```
CREATE TABLE CityMaster(
  Id int PRIMARY KEY,
  City varchar(50) NULL,
  [State] varchar(50) NULL
)

insert into CityMaster
Select 1,'Aurangabad','BIHAR' union all
Select 2,'Aurangabad','MAHARASHTRA' union all
Select 3,'Bijapur','KARNATAKA' union all
Select 4,'Bijapur','CHHATTISGARH' union all
Select 5,'Bilaspur','CHHATTISGARH' union all
Select 6,'Bilaspur','HIMACHAL PRADESH' union all
Select 7,'Jamui','BIHAR' union all
Select 8,'Kullu','HIMACHAL PRADESH' union all
Select 9,'Pune','MAHARASHTRA' union all
```

```sql
Select 10,'Mumbai','MAHARASHTRA' union all
Select 11,'Kolhapur','MAHARASHTRA' union all
Select 12,'Nashik','MAHARASHTRA' union all
Select 13,'Mysore','KARNATAKA' union all
Select 14,'Raigarh','CHHATTISGARH' union all
Select 15,'Aurangabad','BIHAR' union all
Select 16,'Bilaspur','CHHATTISGARH' union all
Select 17,'Bijapur','KARNATAKA'

select * from CityMaster order by city, [state]
```

Anhang

https://www.essentialsql.com/recursive-ctes-explained/ https://www.codeproject.com/Articles/1113586/Recursive-CTEs-Explained

http://blog.revolutionanalytics.com/2015/12/exploring-recursive-ctes-with-sqldf.html

http://www.sqlservercentral.com/articles/T-SQL/90955/

```
SET IDENTITY_INSERT Person ON
GO

INSERT Person (id, name, idVorgesetzter) VALUES (1, 'Don', NULL)
INSERT Person (id, name, idVorgesetzter) VALUES (2, 'Bob', NULL)
INSERT Person (id, name, idVorgesetzter) VALUES (3, 'Niels', 1)
INSERT Person (id, name, idVorgesetzter) VALUES (4, 'Keith', 2)
INSERT Person (id, name, idVorgesetzter) VALUES (5, 'John', 4)
INSERT Person (id, name, idVorgesetzter) VALUES (6, 'Steve', 1)
INSERT Person (id, name, idVorgesetzter) VALUES (7, 'Tim', 2)
INSERT Person (id, name, idVorgesetzter) VALUES (8, 'Peter', 4)
INSERT Person (id, name, idVorgesetzter) VALUES (9, 'Maria', 8)
INSERT Person (id, name, idVorgesetzter) VALUES (10, 'Jaclyn', 3)
INSERT Person (id, name, idVorgesetzter) VALUES (11, 'Anita', 3)

SET IDENTITY_INSERT Person OFF
```

Das Beispiel (*Person*) als gespeicherte Abfrage:

```
CREATE VIEW v_PHierarchisch
as
with PersonH(id, [name], idVorgesetzter, Level) as (
-- this is the anchor member
select id, [name], idVorgesetzter, 0 as Level
  from Person
    where name = 'Bob'

union all

-- this is the recursive member
select e.id, e.[name], e.idVorgesetzter, Level+1
from
  Person e
   join PersonH eh on e.idVorgesetzter = eh.id
)
-- Statement that executes the CTE
select PersonH.id, PersonH.[name], PersonH.idVorgesetzter, Level from PersonH
```

Lösungen

Aufgabe 1

```
with DokumentH(id, Dokumentname, VerknüpftMit) as (
       select id, Dokumentname, VerknüpftMit
         from Dokument
              where Dokumentname='119-Kap.7 _DatenbankL_.pdf'

       union all

       select e.id, e.Dokumentname,e.VerknüpftMit
              from Dokument e
                     join DokumentH eh on e.VerknüpftMit = eh.id

)
select distinct id, Dokumentname, VerknüpftMit
  from DokumentH order by Dokumentname
--select id,VerknüpftMit,Dokumentname from dbo.Dokument where not VerknüpftMit is null
--select id,VerknüpftMit,Dokumentname from dbo.Dokument where id=337

--update Dokument set VerknüpftMit= null where id=337
```

Aufgabe 2

Erstellen sie ein CTE der die ungeraden Zahlen bis 1001 auflistet.

```
WITH cte AS
       (SELECT 1 AS n       -- anchor member

              UNION ALL

              SELECT n+2    -- recursive member
              FROM cte
              WHERE n<1000  -- terminator
       )
SELECT n FROM cte OPTION (MAXRECURSION 1000);
```

Aufgabe 3

http://www.c-sharpcorner.com/UploadFile/d1028b/delete-duplicate-records-from-table-using-cte/
http://www.sqlservercentral.com/blogs/vivekssqlnotes/2013/12/29/deleting-duplicate-rows-using-cte/
http://burnignorance.com/database-tips-and-tricks/how-to-delete-duplicate-rows-in-a-table-using-cte/

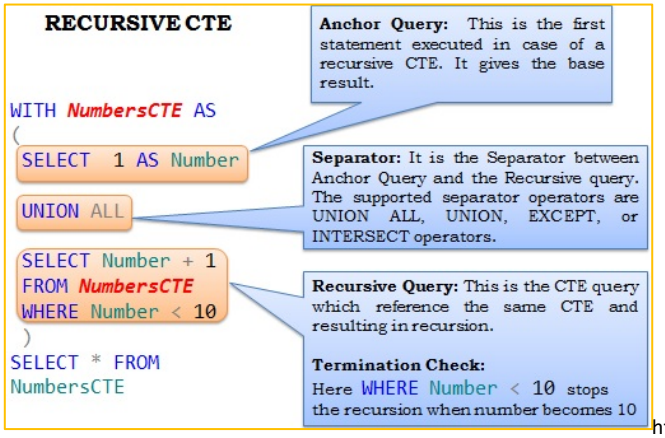
Erstellen sie ein CTE das doppelte Einträge in einer Tabelle löscht.

```
With cte_duplicate ([name], age, salary, rownumber)
as (
  select [name], age, salary, row_number() over(partition by [name], age, salary
    order by [name], age, salary) as rank from tbl_dublicatedata
)
delete from cte_duplicate where rownumber<>1
```

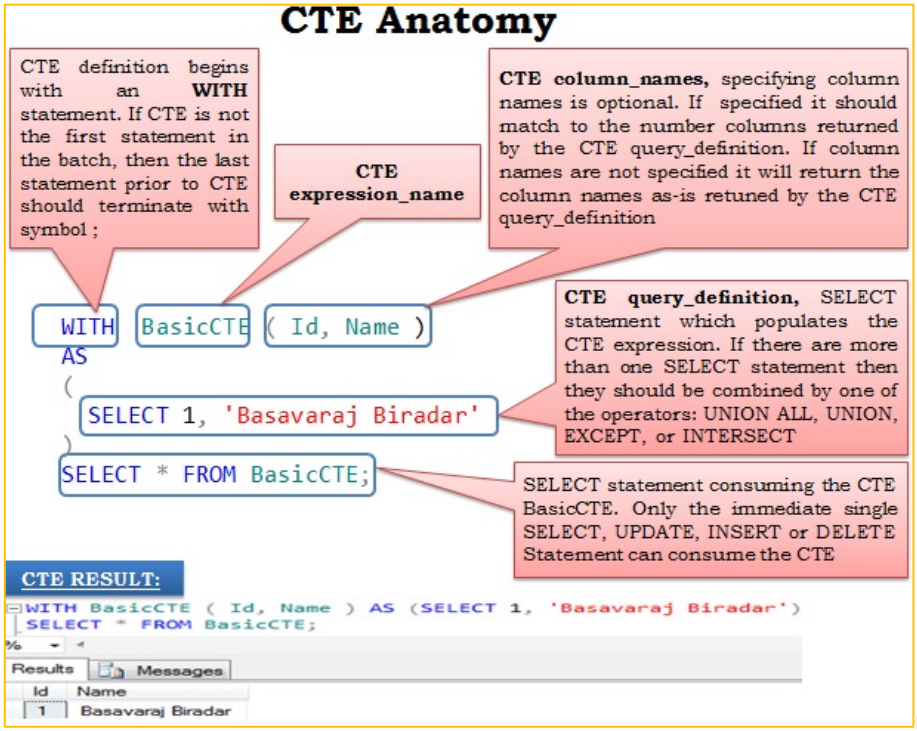
```
With CTE as
(
  Select Id, city, [state], row_number() over (partition by City,[state] order by City) as CityNumber from
       [CityMaster]
)
Select * from CTE order by city, [state]
```

```sql
With CTE as
(
  Select Id, city, [state], row_number() over (partition by City, [state] order by City) as CityNumber from
       [CityMaster]
)
delete from CTE where CityNumber>1
```

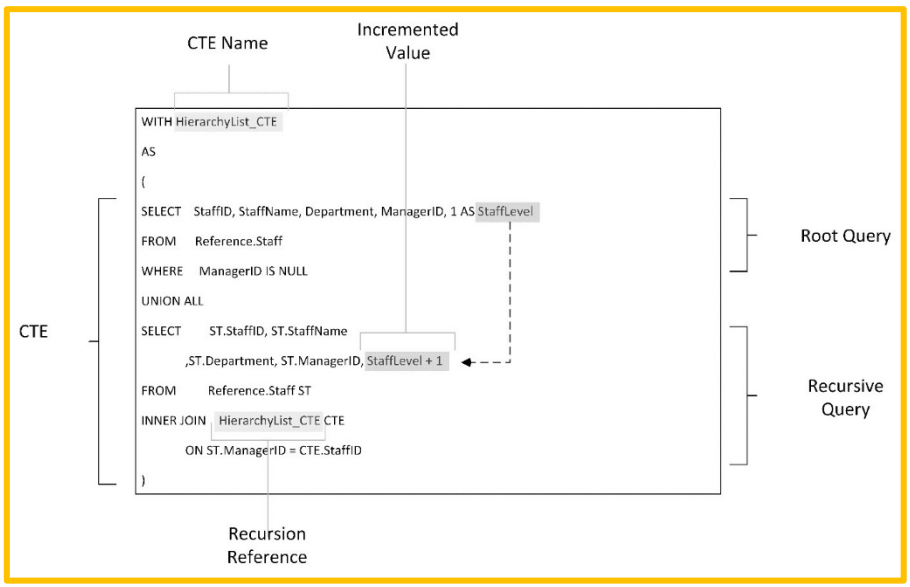
Anhang

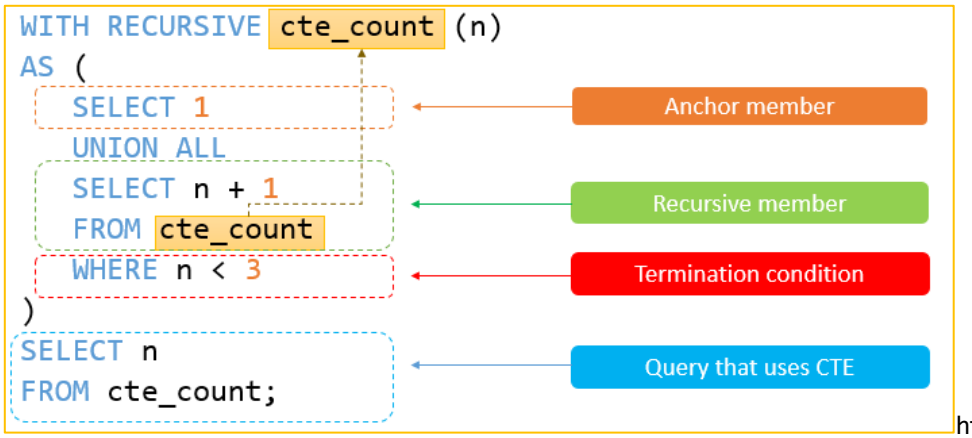


https://sqlhints.com/2015/09/05/recursive-cte-sql-server/

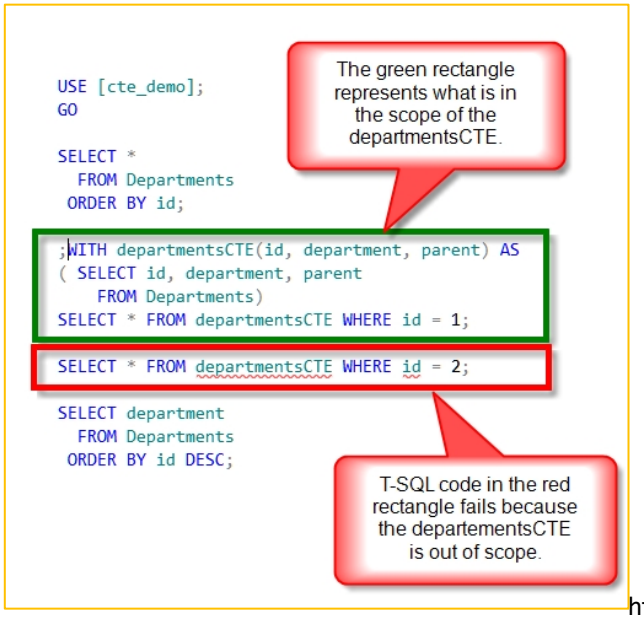


https://sqlhints.com/tag/cte-in-sql-server/





https://www.mysqltutorial.org/mysql-recursive-cte

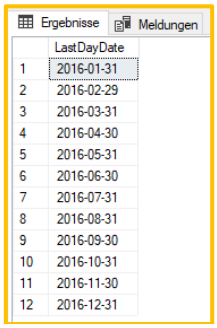


https://stevestedman.com/2013/06/cte-scope/

Beispiel

SQLQueriesSampleData(1).zip

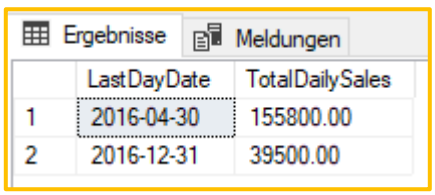
```
;
WITH TallyTable_CTE
AS
(
SELECT ROW_NUMBER() OVER (ORDER BY StockCode) AS ID FROM Data.Stock
)
, LastDayOfMonth_CTE
AS
(
SELECT EOMONTH(DATEFROMPARTS(2016, ID, 1)) AS LastDayDate
FROM TallyTable_CTE WHERE ID <= 12
)
SELECT * FROM LastDayOfMonth_CTE
```



Ergebnisse | Meldungen

	LastDayDate
1	2016-01-31
2	2016-02-29
3	2016-03-31
4	2016-04-30
5	2016-05-31
6	2016-06-30
7	2016-07-31
8	2016-08-31
9	2016-09-30
10	2016-10-31
11	2016-11-30
12	2016-12-31

```
WITH TallyTable_CTE
AS
(
SELECT ROW_NUMBER() OVER (ORDER BY StockCode) AS ID FROM Data.Stock
)
, LastDayOfMonth_CTE
AS
(
SELECT EOMONTH(DATEFROMPARTS(2016, ID, 1)) AS LastDayDate FROM TallyTable_CTE
WHERE ID <= 12
)
SELECT CTE.LastDayDate
             , SUM(SLS.SalePrice) AS TotalDailySales
FROM Data.SalesByCountry SLS
INNER JOIN   LastDayOfMonth_CTE CTE
             ON CTE.LastDayDate = CAST(SLS.SaleDate AS DATE)
GROUP BY CTE.LastDayDate ORDER BY CTE.LastDayDate
```



Ergebnisse | Meldungen

	LastDayDate	TotalDailySales
1	2016-04-30	155800.00
2	2016-12-31	39500.00

Mit mySQL-Syntax

An Introduction to MySQL CTE (mysqltutorial.org)
Common Table Expression (CTE) MySQL 8.0. (wordpress.com)
Recursive Common Table Expressions Overview - MariaDB Knowledge Base
Introduction to MySQL 8.0 Common Table Expressions (Part 1) - Percona Database Performance Blog
Introduction to MySQL 8.0 Recursive Common Table Expression (Part 2) - Percona Database Performance Blog
MySQL :: Other MySQL Documentation

```
WITH RECURSIVE natural_sequence AS
  ( SELECT 1 AS n       -- seed member: our sequence starts from 1
      UNION ALL
    SELECT n+1 FROM natural_sequence    -- recursive member: reference to itself
    WHERE n<10                          -- stop condition
  )
SELECT * FROM natural_sequence;           -- main query
```

```
WITH RECURSIVE factorial(n, fact) AS (
          SELECT 0, 1
          UNION ALL
          SELECT n+1, fact*(n+1)
          FROM factorial
          WHERE n<20)
SELECT * from factorial;
```

```
WITH RECURSIVE fibonacci (n, fib_n, next_fib_n) AS (
          SELECT 1, 0, 1
          UNION ALL
          SELECT n + 1, next_fib_n, fib_n + next_fib_n
          FROM fibonacci
          WHERE n < 20)
SELECT * FROM fibonacci;
```

```
USE db_cte;
with RECURSIVE PersonH (id, Nachname, idVorgesetzter, Level_) as (
       select id, Nachname, idVorgesetzter, null as Level_ from Person where Nachname='Bob'
       union all
       select e.id, e.Nachname, e.idVorgesetzter, Level_
       from Person AS e join PersonH eh on e.idVorgesetzter = eh.id
)
select id, Nachname, idVorgesetzter, Level_ from PersonH
```

```
USE world;
WITH cte (eur_name, eur_population) AS
(SELECT Name, Population FROM country WHERE continent='Europe')
SELECT eur_name, eur_population FROM cte ORDER BY eur_population_ DESC LIMIT 5;
```