

SQLskills Immersion Event

IETS: Advanced Transact-SQL Programming

Module 2: Writing Effective Queries

Bob Beauchemin
BobB@SQLskills.com



Overview

- Query Evaluation
- Inner/Outer Joins (with Search Arguments)
- Query Plans
- Subqueries and Derived Tables
- Set-Based Operations
- Relational Division
- CROSS APPLY and OUTER APPLY
- Common Table Expressions
- Rewriting queries for performance



2

© SQLskills, All rights reserved.
<http://www.SQLskills.com>



Accessing Data – Limit Data

- **Subset of columns = Projection**
 - Do not use *
 - Optimizer has more chances for optimizing query when NARROW
- **Subset of rows = SELECTION**
 - Use positive search arguments
 - Isolate the column to one side of the expression
 - USE: Birthday <= dateadd(yy, -21, getdate())
 - DO NOT USE: dateadd(yy, 21, Birthday) <= getdate()
 - Be cautious with LEADING wildcards
 - USE: LastName LIKE 'S%'
 - Try not to just append %val% to every application
- **Consider using Views, Stored Procedures and Functions to limit the columns/rows**

Table Expressions

- **Joins use table expressions**
- **Table expressions can be:**
 - Permanent Tables
 - Temporary Tables
 - Views
 - Table variables
 - Derived tables
 - Table-valued function
 - Common table expressions
 - Constructed Tables with Row Constructors

Logical Evaluation Order

- **SQL logically evaluated in a specific order**
 - not the order in which the statement is written
 - if optional clause missing, its just skipped
- **Knowing evaluation order helps avoid common coding mistakes**
- **Query processor can reorder the statement in query plan**
 - as long as it doesn't affect results

Evaluation Order

- **Input: 1-n table expressions**
- **Apply table operators**
 - JOIN, CROSS/OUTER APPLY, PIVOT/UNPIVOT
- **JOIN operator**
 - Perform the operator, top to bottom, for n tables
 - Operator handles 2 expressions at a time
 - Add back rows if "OUTER" operation
- **Filter with WHERE**
- **Make groups for GROUP BY**
- **Add extra rows for ROLLUP/CUBE**
- **Filter with HAVING**
- **Generate SELECT list**
 - Filter with DISTINCT
 - Limit with TOP
- **ORDER BY (optional OFFSET and FETCH NEXT)**
- **Apply TOP clause**

Joins

- Introduction to Joins
- Using Inner Joins
- Using Outer Joins
- Using search arguments in a Join
- Joining a table to itself
- Non-equi joins

Join – Terms & Conditions

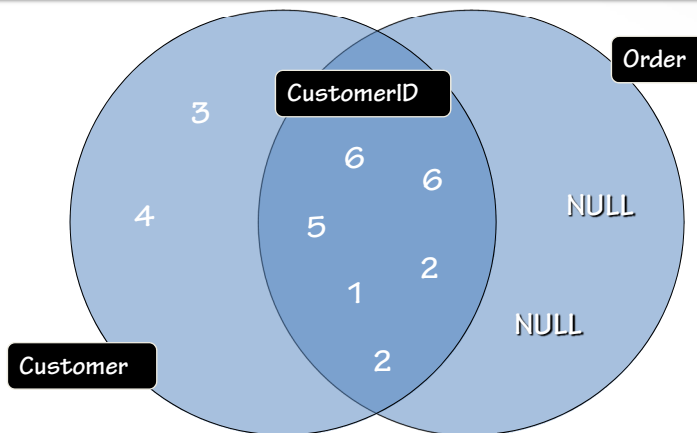
- Inner Join – determines intersection between two or more tables
- Outer Join – adds rows which did not qualify (from the outer table)
- Self Join – Joining a table to itself, used when self-referencing keys exist
- Cross Joins – Usually a mistake ☺ but often for generating sample data or projecting row sets
- Search Arguments – can apply to the entire set or the join condition...

Writing Joins

- **FROM clause first**
 - Establish alias
 - Qualify table name (always use two-part, may need three-part for cross database joins)
- **ON clause second**
 - Establish type of join
 - Define ALL criteria for join
- **Other clauses next**
 - SELECT List
 - WHERE clause
 - ORDER by

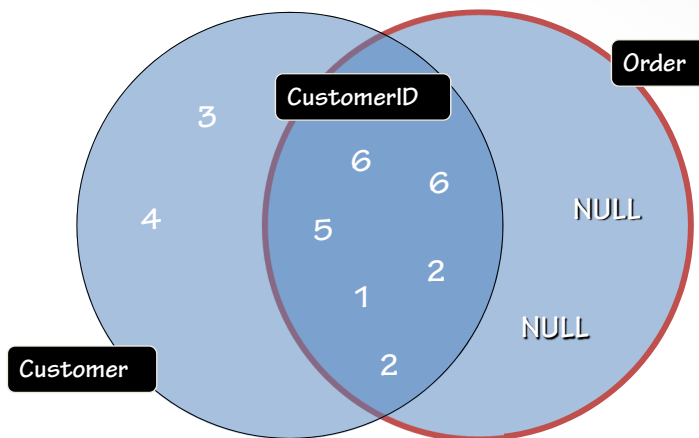
Inner Joins = Intersection

```
SELECT c.CustomerID AS [Customer Number]
      , c.FirstName + ' ' + c.LastName AS [Customer Name]
FROM   dbo.Customer AS c
      INNER JOIN dbo.[Order] AS o
      ON c.CustomerID = o.CustomerID
```



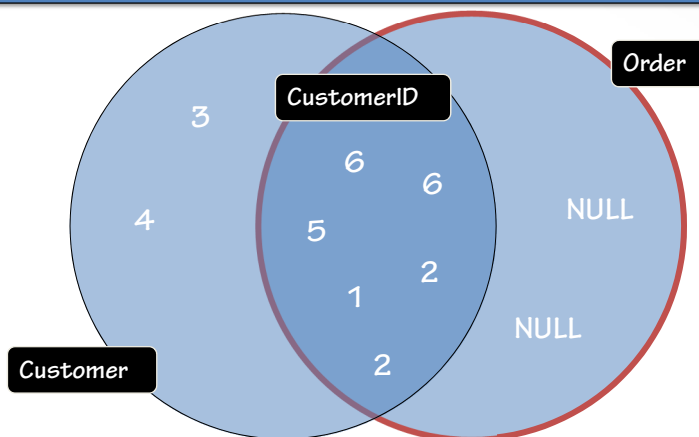
Inner Joins – Not Enough?

What if you want to see ALL orders regardless of whether or not they were purchased by a valid customer?



Outer Joins

```
SELECT c.CustomerID AS [Customer Number]
      , c.FirstName + ' ' + c.LastName AS [Customer Name]
FROM   dbo.Customer AS c
      RIGHT OUTER JOIN dbo.[Order] AS o
      ON c.CustomerID = o.CustomerID
```



Left v. Right – Any difference?

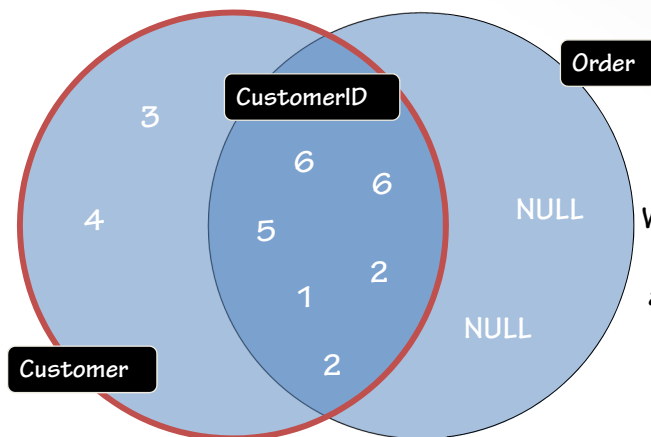
```
SELECT c.CustomerID AS [Customer Number]
      , c.FirstName + ' ' + c.LastName AS [Customer Name]
FROM   dbo.Customer AS c
      RIGHT OUTER JOIN dbo.[Order] AS o
      ON c.CustomerID = o.CustomerID
```

```
SELECT c.CustomerID AS [Customer Number]
      , c.FirstName + ' ' + c.LastName AS [Customer Name]
FROM   dbo.[Order] AS o
      LEFT OUTER JOIN dbo.Customer AS c
      ON c.CustomerID = o.CustomerID
```

- In two table joins – there's no difference between LEFT and RIGHT except for where you place the table within the query (relative to the word JOIN)
- In n-table joins can make a difference in the result set returned
- Personal preference is LEFT

Outer Join

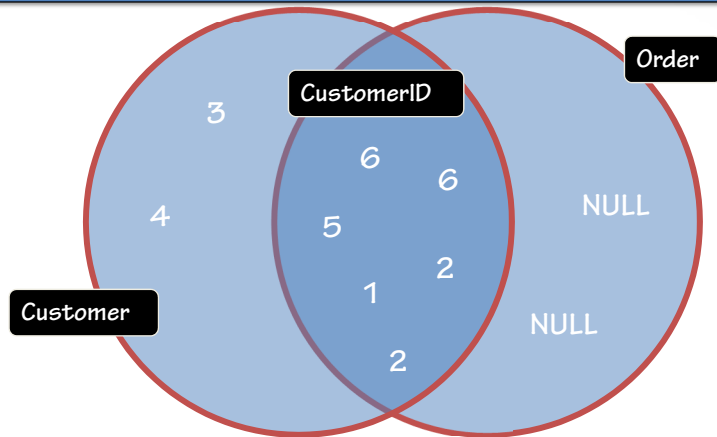
What if you want to see ALL customers regardless of whether or not they placed an order?
Make Customer the OUTER table



What if you want ALL customers and ALL orders?

FULL Outer Join

```
SELECT c.CustomerID AS [Customer Number]
      , c.FirstName + ' ' + c.LastName AS [Customer Name]
FROM   dbo.Customer AS c
      FULL OUTER JOIN dbo.[Order] AS o
      ON c.CustomerID = o.CustomerID
```



Adding Search Arguments

- **Adding a restriction to an INNER Join**
 - Should it apply before or after the join?
 - Does it matter?
 - All conditions MUST be true
- **Should be added to the WHERE clause**

```
SELECT c.CustomerID AS [Customer Number]
      , c.FirstName + ' ' + c.LastName AS [Customer Name]
      , p.ProductName AS [Product Name]
      , o.Quantity
FROM   dbo.Customer AS c
      INNER JOIN dbo.[Order] AS o
      ON c.CustomerID = o.CustomerID
      INNER JOIN dbo.Product AS p
      ON p.ProductID = o.ProductID
      INNER JOIN dbo.Category AS Cat
      ON Cat.CategoryID = p.CategoryID
WHERE cat.CategoryName = 'water-related'
```


Adding Search Arguments

- **Adding a restriction to an OUTER Join**
 - Can be added to FROM clause OR WHERE clause
 - Does it matter? YES!
 - Completely different queries
- **Adding restriction in FROM clause**
 - Inner Join – intersection defined by FROM clause
 - Outer Join – adds rows that did not qualify
- **Adding restriction in WHERE clause**
 - All rows must adhere to this condition

Adding More Tables

- Order is not relevant for performance
- Processing order and internal join type are determined by SQL Server (unless hints used)
- Added table only needs to join to ANY previously listed table
- Watch logic if LEFT/RIGHT/FULL are used – may need to continue adding LEFT/RIGHT/FULL to preserve rows
- Might consider parenthesis to separate LEFT/RIGHT/FULL

Joining a Table to Itself

EmployeeSelfJoin

EmpID	LName	MgrID
1	Smith	NULL
2	Tripp	1

```
SELECT E.FName + ' ' + E.LName AS 'Employee Name',
       M.FName + ' ' + M.LName AS 'Manager Name'
FROM   dbo.EmployeeSelfJoin AS E
       INNER JOIN dbo.EmployeeSelfJoin AS M
       ON E.MgrID = M.EmpID
```

Employee Name	Manager Name
Kimberly Tripp	Bob Smith

What about Bob?

As an "Employee" Bob should show up...

```
-- Use a LEFT OUTER SELF Join!
SELECT E.FName + ' ' + E.LName AS 'Employee Name',
       M.FName + ' ' + M.LName AS 'Manager Name'
FROM   dbo.EmployeeSelfJoin AS E
       LEFT OUTER JOIN dbo.EmployeeSelfJoin AS M
       ON E.MgrID = M.EmpID
```

Employee Name	Manager Name
Bob Smith	NULL
Kimberly Tripp	Bob Smith

Non-Equi Joins

- ON clause not restricted to equals sign
- Range joins can be used to create levels
 - Use BETWEEN in ON clause
- Non-equi joins can be used to create pairs
 - You don't want row to be matched with itself
 - Match is not on foreign key column

Non-Equi Join

```
-- find authors in the same city with related books

SELECT DISTINCT a1.au_fname + ' ' + a1.au_lname AS Author1
, a1.city AS Author1City
, a2.au_fname + ' ' + a2.au_lname AS Author2
FROM authors AS a1
  JOIN titleauthor AS ta1 ON a1.au_id = ta1.au_id
  JOIN titles as t1 ON t1.title_id = ta1.title_id
  JOIN authors as a2 ON a1.city = a2.city
    AND a1.state = a2.state
    AND a1.au_id < a2.au_id
  JOIN titleauthor AS ta2 ON a2.au_id = ta2.au_id
  JOIN titles as t2 ON t2.title_id = ta2.title_id
WHERE t1.type = 'business' AND t2.type = 'business'
```

What is a Query Plan?

- **Query plan is a set of iterators**
 - nested loops/hash match/merge join
 - aggregate iterators
 - others...
- **Two major iterator types**
 - Stop and go - must read all rows before acting on any of them (e.g. sort, hash join)
 - Pipelined - can act on one row at a time
 - (e.g. nested loops, merge join)

Major Iterator Types

- **Three iterator types for joins**
 - Nested Loops
 - Merge
 - Hash
- **Two iterators for aggregation**
 - Stream aggregate
 - Hash aggregate

Parallelism

- **Only considered if multiple processors**
- **Max degree of parallelism decided at exec time**
 - Can be decreased by query hint
- **Three parallelism iterators**
 - Distribute streams
 - Gather streams
 - Repartition streams
- **Five partitioning types**
 - Broadcast
 - Round Robin
 - Hash
 - Range
 - Demand

Subqueries

- Subqueries allow you to use more than query as part of a single SQL statement
- Scalar/Multivalued subqueries
- Standalone/Correlated subqueries
- Subqueries can be used in the SELECT list, FROM, WHERE, ON, HAVING clauses

```
SELECT t.title_id AS [TitleID],  
       t.price - (SELECT AVG(price) FROM titles) AS [PriceDiff]  
FROM   titles AS t
```

Subqueries and Predicates

- You can compare single values or sets
 - In the WHERE clause
 - In the HAVING clause
- Type determines useable predicates
 - Scalar subqueries must be used for scalar predicates
 - Scalar predicates
 - <, >, =, !=, < >, <=, >=
 - Set based (quantified) predicates
 - IN, NOT IN, EXISTS, NOT EXISTS, ANY, ALL

Using Subqueries and Predicates

```
-- single value, OK
-- authors whose books had better than avg sales
SELECT au_id, ta.title_id, title, ytd_sales
  FROM titleauthor AS ta
  JOIN titles t ON ta.title_id = t.title_id
 WHERE ytd_sales >=
        (SELECT AVG(ytd_sales) FROM titles)

-- multivalue, an error
SELECT titles.title_id, title, royalty
  FROM titles
 WHERE titles.royalty >=
        (SELECT roysched.royalty
         FROM roysched)

-- multivalue, OK
-- authors who live in states where there are pubs
SELECT au.au_id, au.au_lname, state
  FROM authors AS au
 WHERE au.state
        IN (SELECT p.state FROM publishers AS p)
```

Subqueries, NOT IN, and NULLs

- Be careful using NOT IN predicates and sets that can contain NULL values
 - NOT IN + set with NULL value = NULL
 - Use ISNULL or COALESCE to be sure

```
-- multivalue, OK
-- authors who live in states where there are not pubs
-- if publisher state contains NULL, no rows are returned
SELECT au.au_id, au.au_lname, state
  FROM authors AS au
 WHERE au.state
        NOT IN (SELECT p.state FROM publishers AS p)
```

Correlated Subqueries

- A correlated subquery is a subquery that refers that a table in the outer query
 - Can also be written as JOIN

```
-- stores that have sold an order of more than 40 books
SELECT s.stor_id, s.stor_name
FROM   stores AS s
WHERE  (SELECT MAX(qty) FROM sales sa
        WHERE sa.stor_id = s.stor_id ) >= 40

SELECT s.stor_id, s.stor_name
FROM   stores AS s
JOIN   sales AS sa ON s.stor_id = sa.stor_id
GROUP BY s.stor_id, s.stor_name
HAVING MAX(qty) >= 40
```

Derived Tables

- A derived table is a subquery that is aliased as a table in the main query
 - Analogous to an "inline view"
 - Useful with GROUP BY
 - Can be used to find duplicate rows

```
-- really simple derived table
SELECT auta.au_fname
      , auta.au_lname
      , auta.title_id
FROM   (
        SELECT au.*, ta.title_id
        FROM   authors AS au
        JOIN   titleauthor AS ta ON au.au_id = ta.au_id
      ) AS auta
```

Relational Operators

- **There are 8 relational operators**

- ❑ Selection (SELECT...WHERE)
- ❑ Projection (SELECT a,b,c... FROM)
- ❑ Join (INNER JOIN)
- ❑ Product (CROSS JOIN)
- ❑ Union (UNION)
- ❑ Intersection (INTERSECT)
- ❑ Difference (EXCEPT)
- ❑ Division

UNION

- **UNION combines the results of multiple queries**

- ❑ UNION ALL keeps duplicates, UNION doesn't
 - ❑ UNION ALL less overhead
 - ❑ UNION ALL used to reconstruct partitioned tables
- ❑ Number of columns must match
- ❑ Data types must be compatible

```
-- combined total sales for both years
SELECT ts.stor_id, SUM(ts.qty)
FROM
(
  SELECT * FROM Sales1993
  UNION ALL
  SELECT * FROM Sales1994
) AS ts
GROUP BY ts.stor_id
```


EXCEPT and INTERSECT

- **SQL Server 2005 added standard set operators**

- INTERSECT is left semi-join
- EXCEPT is left anti semi-join
- SET-based predicates can produce same results
 - WHERE EXISTS ~ INTERSECT
 - WHERE NOT EXISTS ~ EXCEPT
- standard operators enhance compliance

```
-- only authors with books
SELECT au_id FROM authors
INTERSECT
SELECT au_id FROM titleauthor
```

```
-- only authors without books
SELECT au_id FROM authors
EXCEPT
SELECT au_id FROM titleauthor
```

Relational Division

- **Division determines if a value in table A (dividend) is related to every row of corresponding value in table B (divisor)**
 - Exact relational division mandates no "remainder"
- **No division keyword in SQL**
- **Division can be accomplished with a variety of T-SQL constructs**

Two Relational Division Solutions

```
-- Easiest to understand
-- Count languages in list that a person can speak
SELECT P.PersonName, COUNT(*)
FROM   People
WHERE  P.Lang IN (SELECT * FROM Language)
GROUP BY P.PersonName
-- It should be the same number as in the list
HAVING COUNT(*) = (SELECT COUNT(*) FROM Language)

-- Traditional
-- There is no language in the list a person cannot speak
SELECT DISTINCT P1.PersonName FROM Person P1
WHERE NOT EXISTS (
    SELECT L.Lang FROM Languages L
    WHERE NOT EXISTS (
        SELECT P2.Lang FROM Person P2
        WHERE P1.PersonName = P2.PersonName
        AND P2.Lang = L.Lang
    )
)
```

CASE Statement

- **Condition evaluation as part of SELECT**
- **Simple case**
 - CASE keyword is followed by condition
 - Multiple WHEN followed by expression/result
 - Can only compare equality
- **Searched case**
 - CASE keyword has no condition
 - Each WHEN is followed by
 - boolean expression /result
 - Any boolean expression (not just equality)
- **First matching expression produces result**
- **Can have ELSE condition for default**

CASE Examples

```
-- Simple CASE
SELECT ProductNumber, Name,
       Category = CASE ProductLine
                   WHEN 'R' THEN 'Road'
                   WHEN 'M' THEN 'Mountain'
                   WHEN 'T' THEN 'Touring'
                   ELSE 'Not for sale'
                   END
FROM Production.Product
ORDER BY ProductNumber

-- Searched CASE
SELECT ProductNumber, Name,
       'Price Range' = CASE
                       WHEN ListPrice = 0 THEN 'Mfg item - not for resale'
                       WHEN ListPrice < 50 THEN 'Under $50'
                       WHEN ListPrice >= 50 and ListPrice < 250 THEN 'Under $250'
                       ELSE 'Over $1000' END
FROM Production.Product
ORDER BY ProductNumber
```

APPLY

- **APPLY facilitates join operations with table-valued functions**
 - Any statement that returns a rowset can be used
 - Works like a Cartesian product (No ON clause)
- **Allows invoking a table-valued function for each row returned by an outer table expression of a query**
- **The expression on the right side of a CROSS or OUTER APPLY can refer to elements of the expression on the left side**

Using CROSS APPLY

```
-- lowest priced book by publisher
SELECT HT.title AS Title, P.pub_name AS PubName
FROM pubs.dbo.publishers AS P
CROSS APPLY
(SELECT TOP 1 title
 FROM pubs.dbo.titles AS T
 WHERE P.pub_id = T.pub_id
 ORDER BY T.Price
) AS HT

-- this includes publishers who haven't sold a book
SELECT HT.title AS Title, P.pub_name AS PubName
FROM pubs.dbo.publishers AS P
OUTER APPLY
(SELECT TOP 1 title
 FROM pubs.dbo.titles AS T
 WHERE P.pub_id = T.pub_id
 ORDER BY T.Price
) AS HT
```

CROSS APPLY using a UDF

```
CREATE FUNCTION dbo.LeastExpensiveTitle (
    @pub_id CHAR(4)
)
RETURNS TABLE
RETURN
SELECT TOP 1 title
FROM    pubs.dbo.Titles
WHERE   pub_id = @pub_id
ORDER BY Price
```

```
-- use the UDF with CROSS APPLY
SELECT HT.title AS Title, P.pub_name AS PubName
FROM    pubs.dbo.publishers AS P
CROSS APPLY
    dbo.LeastExpensiveTitle(P.pub_id) AS HT
```

OUTER APPLY

- **OUTER APPLY**
 - Returns both rows that produce a result set, and rows that do not
 - Returns NULL values in the columns produced by the table-valued function
- **In the CROSS APPLY example:**
 - This returns publishers with no books

Table Valued Function Usage in Correlated Subquery

- From within a subquery, provides a table-valued function with columns from the outer query as arguments
- Example - Return any publisher whose lowest priced book begins with 'Net'

```
SELECT * FROM Publishers P
WHERE
  (SELECT title
   FROM dbo.LeastExpensiveTitle(P.pub_id))
LIKE 'Net%'
```

Common Table Expression (CTE)

- **Types of CTE**
- **Non-recursive**
 - CTEs which do not have self-references
- **Recursive**
 - CTE's which do have self-references
 - Two or more queries
 - One query must be non-recursive Anchor Member (AM)
 - Other queries are Recursive Members (RM)
 - UNION ALL join
 - All members require exact column match

Table Structure Comparison

- **Derived Table**

```
SELECT * FROM  
(<derived_table_query>) AS  
<dt_alias>
```

- **View**

```
CREATE VIEW <view_name>  
AS  
<view_query>  
GO  
SELECT *  
FROM <view_name>
```

- **CTE**

```
WITH  
<cte_alias>(<column_aliases>)  
AS  
(  
    <cte_query>  
)  
SELECT *  
FROM <cte_alias>
```

Common table expressions

- CTE is a temporary named resultset
 - similar to view, subquery, or derived table
 - can be used with SELECT/INSERT/UPDATE/DELETE

calculates median value

compares to median value

```
WITH mid AS
(
  SELECT ((MAX(value) - MIN(value)) / 2)
    AS midval FROM invoices
)
SELECT
  CASE
    WHEN value > mid.midval THEN 0
    ELSE 1
  END AS half, invoices.*
  FROM invoices, mid
ORDER BY half
```

Common Table Expression Syntax

- Starts with WITH clause
 - aliases specified after CTE name or inline
- Multiple common table expressions, comma separated
 - can reference CTEs defined previously
 - cannot nest

first cte, named low

second cte, named high

SELECT statement

refers to high and low

```
WITH low AS (SELECT ((MAX(value)) / 3)
  AS v FROM invoices),
high AS (SELECT (2 * MAX(value) / 3)
  AS v FROM invoices)
SELECT id, value, value - low.v
  FROM invoices, low, high
 WHERE invoices.value > low.v
  AND invoices.value <= high.v
```

Common Table Expression (CTE)

- CTEs can reference each other and do self-references
- Use the Rule of Referencing Pre-defined CTEs
 - Applies: When one CTE refers to or uses another CTE
 - Rule: only CTEs defined previous to the current CTE can be referenced in the current CTE (applicable to Non-Recursive CTEs)

Multiple CTEs with Referencing

```
WITH YearlyOrderAmtCTE(OrderYear, TotalAmount)
AS
(
    SELECT YEAR(OrderDate), SUM(OrderQty*UnitPrice)
    FROM    Sales.SalesOrderHeader AS H
    JOIN    Sales.SalesOrderDetail AS D
        ON  H.SalesOrderID = D.SalesOrderID
    GROUP BY YEAR(OrderDate)
),
SalesTrendCTE(OrderYear, TotalAmount, AmtYearBefore,
              AmtDifference, PercentDifference)
AS
( -- caution: CTE is evaluated twice
    SELECT ThisYear.OrderYear,
           ThisYear.TotalAmount, LastYear.TotalAmount,
           ThisYear.TotalAmount - LastYear.TotalAmount,
           (ThisYear.TotalAmount/LastYear.TotalAmount - 1) * 100
    FROM    YearlyOrderAmtCTE AS ThisYear
    LEFT OUTER JOIN YearlyOrderAmtCTE AS LastYear
        ON  ThisYear.OrderYear = LastYear.OrderYear + 1
)
SELECT * FROM SalesTrendCTE ORDER BY OrderYear ASC
```


CTE Optimization

- CTE is evaluated only once
 - less scans than subquery if used more than once

evaluated once →

```
WITH low AS (SELECT ((max(value)) / 3)
              AS v FROM invoices),
high AS (SELECT (2 * max(value) / 3)
          AS v FROM invoices)
SELECT id, value, value - low.v
FROM invoices, low, high
WHERE invoices.amount > low.v
AND invoices.value <= high.v
```

evaluated twice →

```
SELECT id, value,
       value - (SELECT (max(value) / 3) FROM invoices)
FROM invoices where
value > (SELECT (max(value) / 3) FROM invoices) and
value < (SELECT (2 * max(value) / 3) FROM invoices)
```

SELECT middle third of invoices



49

© SQLskills. All rights reserved.
<http://www.SQLskills.com>

Recursive CTE

- CTE can do recursive calculation
- Recursive CTE has three parts
 - anchor; precedes UNION ALL; does initialization
 - recursive member after UNION ALL; recurses until no results
 - outer select; selects results to be returned

anchor; executed once →

*recursive member; repeated
joined with previous recursion* →

outer select; ids returned →

```
WITH descendant(parent, id, amount) AS
(SELECT parent, id, amount
 FROM partsTree WHERE id = @start
 UNION ALL
 SELECT P.parent, P.id, P.amount
 FROM partsTree AS P INNER JOIN
 descendant AS A ON P.parent = A.id
 )
SELECT id FROM descendant
```



50

© SQLskills. All rights reserved.
<http://www.SQLskills.com>

How Recursive CTEs Work

- Anchor member is activated. Set R0 (R for Results) is generated
- Recursive member is activated, retrieving set Ri (i = step number) as input to RecursiveCTE, producing Set Ri + 1
- Step 2 is run repeatedly, incrementing the step number each time, until an empty set is returned
- The outer query is executed, retrieving a cumulative (UNION ALL) result of all of the previous steps

Rewriting Queries

- Most non-trivial SQL queries can be written multiple ways
 - JOIN vs correlated subquery
 - JOIN vs derived table
 - MAX or MIN vs ALL
 - NOT IN vs NOT EXISTS
- It's best to rewrite queries multiple ways and test for performance
- Make sure that the answer is the same under all conditions

Comparing Equivalent Queries

- **Compare queries using**
 - SET STATISTICS IO ON
 - SET STATISTICS TIME ON
 - Query Cost
 - Query Plan
- **Ensure you have optimal indexes**
- **Be aware that query engine may get smarter**

Summary

- **Query Evaluation**
- **Joins and Subqueries**
 - Inner joins
 - Outer joins - Left, Right, and Full
 - Cross joins
 - Non-equi joins
- **Query Plans**
- **Set Operators**
- **APPLY Operator**
- **Common Table Expressions**
- **Rewriting queries for performance**

References

- **Statistics Used by the Query Optimizer in Microsoft SQL Server**, Eric Hanson and Lubor Kollar, et al
 - <http://www.microsoft.com/technet/prodtechnol/sql/2005/qrystats.mspx>
- **Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server**, Arun Marathe, et al
 - <http://www.microsoft.com/technet/prodtechnol/sql/2005/recomp.mspx>

Questions?