

CTEs, Views, Temporary Tables, Stored Procedures & Functions

Write queries other people can read and reuse. Use CTEs (including recursive CTEs for hierarchies and calendars), create views for reusable logic and security, use temp tables and table variables, and build stored procedures, user-defined functions, parameters and basic error handling in T-SQL.

● SQL class 6

✦ SQL Server 2022, 2026

Advanced

🕒 130 min

By the end of this lesson you can

- Structure complex queries with one or more CTEs and explain CTE vs subquery vs temp table
- Write recursive CTEs for hierarchies (org charts) and number or date series
- Create, alter and use views, and explain what views are good for
- Choose between temp tables, table variables and CTEs
- Write stored procedures with parameters, scalar and table-valued functions, and TRY...CATCH

Datasets used

superstore_sales.csv

hr_employees.csv

CTE: a named, readable subquery

A **Common Table Expression** is a named result set defined with `WITH` that exists for **one statement**. It makes complex logic read top to bottom, like paragraphs, instead of inside-out.

SQL

```
WITH name1 AS (  
    SELECT ...  
)  
name2 AS (  
    SELECT ... FROM name1 ...      -- later CTEs can use earlier ones  
)  
SELECT ... FROM name2 ...;
```

From nested subqueries to CTEs

Question: "Which regions had above-average 2025 sales per customer?"

SQL

```
WITH customer_sales AS (                                -- step 1: sales per customer in 2025
    SELECT c.region, c.customer_id, SUM(oi.sales) AS sales
    FROM order_items AS oi
    JOIN orders AS o ON o.order_id = oi.order_id
    JOIN customers AS c ON c.customer_id = o.customer_id
    WHERE o.order_date ≥ '2025-01-01' AND o.order_date < '2026-01-01'
    GROUP BY c.region, c.customer_id
),
region_avg AS (                                         -- step 2: average per region
    SELECT region, COUNT(*) AS customers, AVG(sales) AS avg_sales_per_customer
    FROM customer_sales
    GROUP BY region
)
SELECT region, customers,
    ROUND(avg_sales_per_customer, 0) AS avg_sales_per_customer,
    ROUND((SELECT AVG(sales) FROM customer_sales), 0) AS overall_avg
FROM region_avg
WHERE avg_sales_per_customer > (SELECT AVG(sales) FROM customer_sales) -- step 3: compare
ORDER BY avg_sales_per_customer DESC;
```

RESULT

region	customers	avg_sales_per_customer	overall_avg
West	156	142,513	125,805
South	160	130,010	125,805

2 rows × 4 columns

customer_sales is used **twice**, which with plain subqueries would mean copy-pasting the same block. CTEs also make window-function filters (Lesson 11) clean.

TIP CTE style

Name CTEs for what they contain (customer_sales, not cte1); keep each CTE doing one job; comment each step. If a CTE needs a long explanation, it probably should be two CTEs.

CTE, subquery, view and temp table

	Subquery / derived table	CTE	View	Temp table
Lives for	The statement	The statement	Permanently (a saved query)	The session (#t)
Stores data?	No	No (re-evaluated)	No (unless indexed/materialised)	Yes
Reusable in other queries	No	No	Yes	In the same session
Can be referenced several times	Only by repeating	Yes , in the same statement	Yes	Yes
Recursive	No	Yes	No	No
Can be indexed	No	No	Indexed views only	Yes
Best for	Simple one-off logic	Readability, multi-step logic, recursion	Sharing logic, hiding complexity, security	Large intermediate results used many times

NOTE A CTE is not a cache

In SQL Server the CTE's query may run each time it is referenced. If an expensive CTE is used many times, load it into a **temp table** once.

Recursive CTEs

A recursive CTE has an **anchor** (starting rows) and a **recursive member** that refers to the CTE itself, joined with UNION ALL. It repeats until no new rows are produced.

Number and date series

SQL

```
WITH numbers AS (
    SELECT 1 AS n                                -- anchor
    UNION ALL
    SELECT n + 1 FROM numbers WHERE n < 7      -- recursive member with a stop condition
)
SELECT n FROM numbers;
```

RESULT

n
1
2
3
4
5
6
7

7 rows × 1 columns

A **calendar table** of every day in 2026, handy for filling gaps before LAG/LEAD (Lesson 12):

SQL

```
WITH calendar AS (  
    SELECT CAST('2026-01-01' AS DATE) AS d  
    UNION ALL  
    SELECT CAST(DATEADD(day, 1, d) AS DATE) FROM calendar WHERE d < CAST('2026-12-31' AS  
DATE)  
)  
SELECT COUNT(*) AS days, MIN(d) AS first_day, MAX(d) AS last_day,  
       SUM(CASE WHEN DATENAME(weekday, d) IN ('Saturday', 'Sunday') THEN 1 ELSE 0 END) AS  
weekend_days  
FROM calendar;
```

RESULT

days	first_day	last_day	weekend_days
365	2026-01-01	2026-12-31	104

1 rows × 4 columns

In SQL Server, the default limit is 100 recursions, so a year-long calendar needs `OPTION (MAXRECURSION 400)` after the final `SELECT` (use 0 for no limit, with care).

Hierarchies: org chart

SQL

```
CREATE TABLE org (emp_id INT, name VARCHAR(20), manager_id INT);
INSERT INTO org VALUES (1,'Varun',NULL), (2,'Asha',1), (3,'Ravi',1), (4,'Meera',2),
(5,'Kabir',2), (6,'Zoya',4), (7,'Farhan',3);

WITH tree AS (
    SELECT emp_id, name, manager_id, 0 AS level, CAST(name AS VARCHAR(200)) AS path
    FROM org WHERE manager_id IS NULL -- anchor: the top
    UNION ALL
    SELECT o.emp_id, o.name, o.manager_id, t.level + 1, CAST(CONCAT(t.path, ' > ', o.name)
AS VARCHAR(200))
    FROM org AS o
    JOIN tree AS t ON o.manager_id = t.emp_id -- walk down one level
)
SELECT level, CONCAT(REPLICATE(' ', level), name) AS org_chart, path
FROM tree
ORDER BY path;
```

RESULT

Commands completed successfully.
(7 rows affected)

level	org_chart	path
0	Varun	Varun
1	Asha	Varun > Asha
2	Kabir	Varun > Asha > Kabir
2	Meera	Varun > Asha > Meera
3	Zoya	Varun > Asha > Meera > Zoya
1	Ravi	Varun > Ravi
2	Farhan	Varun > Ravi > Farhan

7 rows × 3 columns

Recursive CTEs also solve bill-of-materials, category trees and graph-path problems.

Views

A **view** is a saved SELECT with a name. Querying it runs the stored query; it holds no data of its own.

SQL

```
CREATE VIEW vw_order_detail AS
SELECT o.order_id, o.order_date, o.ship_mode,
       c.customer_id, c.customer_name, c.segment, c.city, c.region,
       p.product_id, p.product_name, p.category, p.sub_category,
       oi.quantity, oi.discount, oi.sales, oi.profit
FROM order_items AS oi
JOIN orders      AS o ON o.order_id    = oi.order_id
JOIN customers   AS c ON c.customer_id = o.customer_id
JOIN products    AS p ON p.product_id  = oi.product_id;

SELECT category, region, ROUND(SUM(sales), 0) AS sales
FROM vw_order_detail
WHERE order_date ≥ '2025-01-01'
GROUP BY category, region
ORDER BY category, region;
```

RESULT

category	region	sales
Furniture	East	2,875,816
Furniture	North	2,688,426
Furniture	South	3,759,688
Furniture	West	3,436,316
Office Supplies	East	1,075,076
Office Supplies	North	833,204
Office Supplies	South	1,018,343
Office Supplies	West	1,044,699
Technology	East	14,959,309
Technology	North	12,911,446
Technology	South	16,023,523
Technology	West	17,750,953

12 rows × 3 columns

Now analysts query one friendly object instead of repeating a four-table join. Power BI and Excel can connect to the view too.

Use views to	Notes
Hide complexity	One name for a multi-join query
Provide one definition of a metric	"Net sales" defined once, used everywhere
Secure data	GRANT SELECT ON vw_sales_summary TO analysts without granting access to base tables; expose only some columns/rows
Keep apps working when tables change	Change the view, not every report

SQL Server syntax: CREATE VIEW dbo.vw_x AS ..., ALTER VIEW, DROP VIEW IF EXISTS, CREATE OR ALTER VIEW. Rules: no ORDER BY inside a view (unless with TOP); views can usually be INSERT/UPDATED only if they map simply to one table. **Indexed (materialised) views** store the results physically for speed (WITH SCHEMABINDING + a unique clustered index).

Temporary tables and table variables

SQL

```
-- Temp table: lives for the session, stored in tempdb, can be indexed, has statistics
SELECT customer_id, SUM(sales) AS sales
INTO #cust_sales -- the # prefix makes it local temporary
FROM superstore
GROUP BY customer_id;

CREATE INDEX ix_cust ON #cust_sales (customer_id);
SELECT TOP 10 * FROM #cust_sales ORDER BY sales DESC;
DROP TABLE IF EXISTS #cust_sales;

-- Table variable: scoped to the batch; for small row counts
DECLARE @top TABLE (customer_id VARCHAR(10), sales DECIMAL(14,2));
INSERT INTO @top SELECT TOP 5 customer_id, SUM(sales) FROM superstore GROUP BY customer_id
ORDER BY SUM(sales) DESC;
SELECT * FROM @top;
```

##name is a **global** temp table visible to other sessions. Temp tables vanish when the session ends.

Stored procedures

A **stored procedure** is a named, saved block of T-SQL that can take parameters, run many statements, and return results. It is compiled and its plan cached, and permissions can be granted on the procedure instead of the tables.

SQL

```
CREATE OR ALTER PROCEDURE dbo.usp_sales_by_region
    @from_date DATE,
    @to_date DATE,
    @category VARCHAR(30) = NULL          -- optional parameter with a default
AS
BEGIN
    SET NOCOUNT ON;                    -- suppress "(n rows affected)" messages

    SELECT c.region,
           COUNT(DISTINCT o.order_id) AS orders,
           SUM(oi.sales)              AS sales,
           SUM(oi.profit)             AS profit
    FROM   dbo.order_items AS oi
    JOIN   dbo.orders      AS o ON o.order_id   = oi.order_id
    JOIN   dbo.customers   AS c ON c.customer_id = o.customer_id
    JOIN   dbo.products    AS p ON p.product_id = oi.product_id
    WHERE  o.order_date ≥ @from_date
           AND o.order_date < DATEADD(day, 1, @to_date)
           AND (@category IS NULL OR p.category = @category)    -- optional filter
    GROUP BY c.region
    ORDER BY sales DESC;
END;
GO

EXEC dbo.usp_sales_by_region @from_date = '2025-01-01', @to_date = '2025-12-31';
EXEC dbo.usp_sales_by_region '2025-01-01', '2025-12-31', 'Technology';
```

Output parameters, variables, IF and loops

SQL

```
CREATE OR ALTER PROCEDURE dbo.usp_add_order_note
    @order_id VARCHAR(20),
    @note      NVARCHAR(200),
    @rows      INT OUTPUT
AS
BEGIN
    SET NOCOUNT ON;
    IF NOT EXISTS (SELECT 1 FROM dbo.orders WHERE order_id = @order_id)
    BEGIN
        RAISERROR('Order %s does not exist.', 16, 1, @order_id); -- or THROW 50001,
        'Order does not exist.', 1;
        RETURN;
    END;

    UPDATE dbo.orders SET note = @note WHERE order_id = @order_id;
    SET @rows = @@ROWCOUNT; -- rows changed by the
    last statement
END;
GO

DECLARE @n INT;
EXEC dbo.usp_add_order_note 'IN-2025-00042', N'Gift wrap', @rows = @n OUTPUT;
SELECT @n AS rows_updated;
```

Control flow	Syntax
Condition	IF ... BEGIN ... END ELSE BEGIN ... END
Loop	WHILE @i ≤ 10 BEGIN ... SET @i += 1; END
Exit / return	RETURN
Row count	@@ROWCOUNT
Dynamic SQL	EXEC sp_executesql @sql, N'@x INT', @x = 5 (use parameters; never concatenate user input: SQL injection)

Error handling: TRY...CATCH

SQL

```
BEGIN TRY
    BEGIN TRAN;
        UPDATE dbo.accounts SET balance = balance - 5000 WHERE acct_id = 1;
        UPDATE dbo.accounts SET balance = balance + 5000 WHERE acct_id = 2;
    COMMIT;
END TRY
BEGIN CATCH
    IF @@TRANCOUNT > 0 ROLLBACK;
    SELECT ERROR_NUMBER() AS err_no, ERROR_MESSAGE() AS err_msg, ERROR_LINE() AS err_line;
    THROW;
    -- re-raise to the caller
END CATCH;
```

User-defined functions (UDFs)

Type	Returns	Example use
Scalar	One value	dbo.fn_margin(profit, sales)
Inline table-valued	A table, from one SELECT	dbo.fn_orders_by_city('Pune'): like a parameterised view
Multi-statement table-valued	A table built in a body	Complex logic (usually slower)

SQL

```
CREATE OR ALTER FUNCTION dbo.fn_margin (@profit DECIMAL(12,2), @sales DECIMAL(12,2))
RETURNS DECIMAL(6,2)
AS
BEGIN
    RETURN CASE WHEN @sales = 0 THEN NULL ELSE 100.0 * @profit / @sales END;
END;
GO

CREATE OR ALTER FUNCTION dbo.fn_sales_by_city (@city VARCHAR(30))
RETURNS TABLE
AS RETURN (
    SELECT order_id, order_date, sales, profit FROM dbo.vw_order_detail WHERE city = @city
);
GO

SELECT order_id, dbo.fn_margin(profit, sales) AS margin_pct FROM
dbo.fn_sales_by_city('Pune');
```

WATCH OUT Scalar UDFs can be slow

Older SQL Server versions call a scalar UDF once per row, which can make a query many times slower than writing the expression inline (SQL Server 2019+ can inline some). Prefer inline table-valued functions or views for set-based logic.

Procedure vs function vs view

	View	Stored procedure	Function
Parameters	No	Yes	Yes
Can change data (INSERT/UPDATE)	Limited	Yes	No (side-effect free)
Used in SELECT / FROM	Yes	No (EXEC)	Yes
Returns	A table	Result sets, output parameters, a status code	A value or a table

Triggers (awareness)

A **trigger** runs automatically on INSERT, UPDATE or DELETE: audit logs, maintaining summary tables. Use sparingly: they hide logic and can slow writes. Modern designs prefer explicit procedures, constraints, temporal tables or change data capture.

DO IT WITH AI Do it with AI

Prompt: *"Convert this nested-subquery SQL into readable CTEs with clear names and comments, keep the result identical, and explain each step."* AI is good at refactoring; verify equality by running both versions and comparing with EXCEPT in both directions (Lesson 9).

Summary

KEY TAKEAWAYS

- **CTE** (`WITH name AS (...)`): a named, readable subquery for one statement; can be referenced many times; **recursive CTEs** generate series and walk hierarchies (anchor + `UNION ALL` + recursive member).
- **View**: a saved query for reuse, one definition of a metric, and security. **Temp tables** (`#t`) store intermediate data and can be indexed; table variables suit small sets.
- **Stored procedures** take parameters, run multiple statements, change data and handle errors with `TRY...CATCH`; **functions** return values or tables and can be used inside queries.
- Prefer set-based SQL; avoid row-by-row loops and scalar UDFs on big tables; never concatenate user input into dynamic SQL.

Quick quiz

6 questions · tap an answer to check it

1. A CTE exists for:

A The whole session

B A single statement

C Until the server restarts

D As long as the database

Answer: B. A CTE is defined and used within one statement.

2. What are the two parts of a recursive CTE?

- A SELECT and WHERE
- B An anchor member and a recursive member joined with UNION ALL**
- C INSERT and UPDATE
- D A view and a function

Answer: B. The anchor gives starting rows; the recursive member repeats until nothing new appears.

3. Which object stores data physically for the session and can be indexed?

- A CTE
- B View
- C Temporary table**
- D Subquery

Answer: C.

temp tables live in tempdb for the session.

4. Which can modify data and return an OUTPUT parameter?

- A View
- B Stored procedure**
- C Scalar function
- D CTE

Answer: B. Procedures can run INSERT/UPDATE/DELETE and return output parameters.

5. Why grant access to a view instead of the base table?

- A** Views are faster
- B** You can expose only selected columns and rows and hide the underlying tables
- C** Views cannot be dropped
- D** Tables cannot be queried

Answer: B. Views are a security and simplification layer.

6. A recursive CTE stops with an error after 100 levels. Fix?

- A** Add DISTINCT
- B** Add OPTION (MAXRECURSION n) (and a proper stop condition)
- C** Use a view
- D** Use TOP

Answer: B. SQL Server's default recursion limit is 100.

EXERCISE Build reusable objects

(1) A view `vw_monthly_sales` with year, month, category and total sales. (2) A CTE query that lists, for each region, the best-selling sub-category and its share of the region's sales. (3) A stored procedure `usp_top_customers` (@year INT, @n INT) that returns the top n customers by sales for a year. (4) A recursive CTE that lists all employees under manager 2 in the `org` table with their level.

Show solution (worked answer)

SQL

```
-- (1)
CREATE VIEW vw_monthly_sales AS
SELECT YEAR(order_date) AS yr, MONTH(order_date) AS mth, category, SUM(sales) AS sales
FROM vw_order_detail GROUP BY YEAR(order_date), MONTH(order_date), category;

-- (2)
WITH rs AS (SELECT region, sub_category, SUM(sales) AS sales FROM superstore GROUP BY
region, sub_category),
    r AS (SELECT *, RANK() OVER (PARTITION BY region ORDER BY sales DESC) AS rnk,
100.0*sales/SUM(sales) OVER (PARTITION BY region) AS pct FROM rs)
SELECT region, sub_category, ROUND(pct,1) AS pct_of_region FROM r WHERE rnk = 1;

-- (3) T-SQL
CREATE OR ALTER PROCEDURE usp_top_customers @year INT, @n INT AS
BEGIN
    SELECT TOP (@n) customer_id, customer_name, SUM(sales) AS sales FROM vw_order_detail
WHERE YEAR(order_date) = @year
    GROUP BY customer_id, customer_name ORDER BY sales DESC;
END;

-- (4)
WITH t AS (SELECT emp_id, name, 1 AS level FROM org WHERE manager_id = 2
    UNION ALL SELECT o.emp_id, o.name, t.level + 1 FROM org o JOIN t ON
o.manager_id = t.emp_id)
SELECT * FROM t;
```

INTERVIEW CORNER

Questions you may be asked

Q: CTE vs temp table vs subquery? Subquery: inline, single use. CTE: named, readable, statement-scoped, supports recursion, may be re-evaluated. Temp table: materialised data for the session, indexable: use for large intermediate results reused many times.

Q: View vs stored procedure? A view is a saved SELECT you query like a table; a procedure is a program with parameters and multiple statements that can change data.

Q: What is a recursive CTE and where do you use it? A CTE that references itself, for hierarchies (org charts, categories), series generation and graph traversal.

Q: How do you prevent SQL injection in dynamic SQL? Use parameterised `sp_executesql` and `QUOTENAME` for identifiers; never concatenate raw user input.

Practise more online

Free, trusted places to test yourself on this topic. Links open the original site in a new tab.