SQL Tutorial for Beginners

Complete Course Notes — Basics, Intermediate & Advanced SQL

SECTION 1: Setup — Getting Started with SQL Server

1.1 What You Need to Download

To follow this course you need two downloads, both available from Microsoft (links in the course description):

- SQL Server (the database engine) — download the free Express edition, which is smaller and sufficient for learning
- SQL Server Management Studio (SSMS) — the graphical interface used to write and run SQL queries

Once SQL Server is installed and SSMS is open, connect to your local SQL Express server. The connection is automatic for a local install — simply click Connect.

1.2 Creating Your First Database and Tables

Everything in SQL is organised inside a database. The first step is to create one:

- Right-click Databases in the Object Explorer → New Database → name it SQLTutorial
- Use a New Query window (not the GUI table designer) to create tables using T-SQL scripts

The two tables used throughout this course:

EmployeeDemographics table:

```
CREATE TABLE EmployeeDemographics
(
  EmployeeID   INT,
  FirstName    VARCHAR(50),
  LastName     VARCHAR(50),
  Age          INT,
  Gender       VARCHAR(50)
)
```

EmployeeSalary table:

```
CREATE TABLE EmployeeSalary
(
  EmployeeID   INT,
  JobTitle     VARCHAR(50),
  Salary       INT
)
```

1.3 Inserting Data

Once tables are created, populate them with INSERT INTO statements:

```
INSERT INTO EmployeeDemographics VALUES
(101, 'Jim', 'Halpert', 30, 'Male'),
(102, 'Pam', 'Beasley', 28, 'Female'),
(103, 'Dwight', 'Schrute', 29, 'Male')
```

- INSERT INTO creates a new row — it does not modify existing rows
- Columns must be given values in the order they were defined, unless you specify column names explicitly
- Large inserts can be grouped into a single VALUES list, or the script can be downloaded from GitHub and copy-pasted

1.4 Key SQL Concepts: Databases, Schemas, and Tables

Concept	Explanation
Database	The top-level container for all tables, views, procedures, and data. You must be connected to the correct database.
Schema (dbo)	A namespace within a database. The default schema is dbo (database owner). Full table reference: DatabaseName.dbo.TableName
Table	A structured set of rows and columns — the primary unit of data storage in SQL.
Column	A named attribute of a table with a defined data type (INT, VARCHAR, DATE, etc.).
Row / Record	A single entry in a table representing one instance of the entity.
NULL	The absence of a value — not zero, not empty string. Needs special handling (IS NULL / IS NOT NULL).

SECTION 2: The Basics — SELECT, FROM, and WHERE

2.1 SELECT and FROM

SELECT and FROM are the two most fundamental SQL keywords. Every query starts with them.

Syntax / Feature	Example & Explanation
Select all columns	SELECT * FROM EmployeeDemographics — the star (*) returns every column and every row
Select specific columns	SELECT FirstName, LastName FROM EmployeeDemographics — returns only the named columns
TOP N rows	SELECT TOP 5 * FROM EmployeeDemographics — limits result to first 5 rows; useful for previewing large tables
Full table reference	SELECT * FROM SQLTutorial.dbo.EmployeeSalary — explicitly specifies database, schema, and table; works regardless of which database is selected in SSMS
Column alias (AS)	SELECT COUNT(LastName) AS LastNameCount FROM EmployeeDemographics — renames the output column. AS is optional — a space also works

2.2 DISTINCT — Returning Unique Values

DISTINCT filters the result to return only unique values in the specified column(s). It is similar to GROUP BY but simpler — it just shows what unique values exist without aggregating.

```
SELECT DISTINCT Gender FROM EmployeeDemographics
```

- Returns: Male, Female — only the distinct values in the Gender column
- SELECT DISTINCT EmployeeID returns all 9 rows because every ID is already unique
- DISTINCT on multiple columns returns rows where the combination of those columns is unique

2.3 Aggregate Functions

Aggregate functions perform a calculation on a set of rows and return a single result. They are used in SELECT statements and are the foundation of data analysis in SQL.

Function	What It Does
COUNT(column)	Counts non-NULL values in the column
COUNT(*)	Counts all rows including NULLs
MAX(column)	Returns the largest value
MIN(column)	Returns the smallest value
AVG(column)	Returns the arithmetic mean
SUM(column)	Returns the total of all values

Important: NULLs are ignored by all aggregate functions except COUNT(*). If a salary value is NULL it is excluded from AVG, SUM, and MAX calculations.

2.4 WHERE — Filtering Rows

The WHERE clause filters which rows are returned. It evaluates a condition for each row and only includes rows where the condition is TRUE.

```
SELECT * FROM EmployeeDemographics WHERE Age > 30
SELECT * FROM EmployeeDemographics WHERE Gender = 'Male'
SELECT * FROM EmployeeDemographics WHERE LastName = 'Scott' AND Age > 25
SELECT * FROM EmployeeDemographics WHERE Gender = 'Male' OR Age < 25
```

Operator	Meaning
=	Equal to
<> or !=	Not equal to
> / <	Greater than / Less than
>= / <=	Greater than or equal / Less than or equal
AND	Both conditions must be true
OR	At least one condition must be true
LIKE	Pattern matching with wildcards
IN	Matches any value in a list
IS NULL	Matches NULL values
IS NOT NULL	Matches non-NULL values
BETWEEN	Inclusive range

SECTION 3: GROUP BY and ORDER BY

3.1 GROUP BY — Aggregating Data

GROUP BY groups rows that share the same value in one or more columns into summary rows. It is used with aggregate functions to compute per-group statistics. Every column in the SELECT list must either be inside an aggregate function OR appear in the GROUP BY clause.

```
SELECT Gender, COUNT(Gender) AS GenderCount
FROM EmployeeDemographics
GROUP BY Gender
```

- Result: one row for Male (showing count of 6) and one row for Female (showing count of 3)
- DISTINCT just shows unique values; GROUP BY rolls all matching rows into one and allows aggregate calculations on that group
- Multiple columns in GROUP BY: each unique combination of those columns forms its own group

```
SELECT Gender, Age, COUNT(Gender) AS GenderCount
FROM EmployeeDemographics
GROUP BY Gender, Age
```

- WHERE can still be used with GROUP BY — it filters rows BEFORE grouping happens:

```
SELECT Gender, COUNT(Gender) AS GenderCount
FROM EmployeeDemographics
WHERE Age > 31
GROUP BY Gender
```

3.2 ORDER BY — Sorting Results

ORDER BY sorts the query result by one or more columns. By default the sort is ascending (smallest to largest, A to Z). Add DESC for descending order.

```
SELECT * FROM EmployeeDemographics ORDER BY Age ASC
SELECT * FROM EmployeeDemographics ORDER BY Age DESC
```

- Multiple sort columns — primary sort first, then secondary:

```
SELECT * FROM EmployeeDemographics ORDER BY Age DESC, Gender ASC
```

- You can reference columns by their position number instead of name:

```
ORDER BY 4 DESC, 5 ASC
```

  - 4 refers to the 4th column in the SELECT list (Age), 5 refers to the 5th (Gender)
- ORDER BY can reference columns not in the SELECT list
- Default is ascending — ASC keyword is optional

3.3 HAVING — Filtering After Aggregation

HAVING is a filter applied AFTER the GROUP BY aggregation. It is used because WHERE cannot filter on aggregate function results — WHERE runs before aggregation, HAVING runs after.

Rule: if you want to filter on COUNT(), AVG(), SUM(), MAX(), or MIN() — use HAVING, not WHERE.

```
SELECT JobTitle, COUNT(JobTitle) AS JobCount
FROM EmployeeSalary
GROUP BY JobTitle
HAVING COUNT(JobTitle) > 1
```

- Returns only job titles held by more than one employee — the COUNT cannot be used in WHERE

```
SELECT JobTitle, AVG(Salary) AS AvgSalary
FROM EmployeeSalary
GROUP BY JobTitle
HAVING AVG(Salary) > 45000
ORDER BY AVG(Salary) DESC
```

- Returns only job titles where the average salary exceeds $45,000, ordered highest to lowest

Full SQL clause order — this is the order clauses must be written AND the order in which SQL executes them:

Order	Clause
1	SELECT
2	FROM
3	JOIN ... ON
4	WHERE
5	GROUP BY
6	HAVING
7	ORDER BY

SECTION 4: Modifying Data — UPDATE and DELETE

4.1 UPDATE — Modifying Existing Rows

UPDATE modifies the value of one or more columns in existing rows. Unlike INSERT, it does not create new rows — it changes rows that already exist.

```
UPDATE SQLTutorial.dbo.EmployeeDemographics
SET EmployeeID = 112,
    Age = 31,
    Gender = 'Female'
WHERE FirstName = 'Holly' AND LastName = 'Flax'
```

- SET specifies the column(s) and their new value(s) — separate multiple column updates with commas
- WHERE restricts the update to specific rows — without WHERE, EVERY row in the table will be updated
- Always include a WHERE clause unless you intentionally want to update every row
- Use a unique key (e.g. EmployeeID) in WHERE for precision:

```
UPDATE EmployeeDemographics SET Age = 31 WHERE EmployeeID = 102
```

4.2 DELETE — Removing Rows

DELETE removes entire rows from a table. This action is permanent and cannot be undone — there is no built-in undo for a DELETE statement.

```
DELETE FROM SQLTutorial.dbo.EmployeeDemographics
WHERE EmployeeID = 105
```

- Without WHERE: DELETE FROM EmployeeDemographics deletes EVERY row in the table — the table structure remains but is now empty
- This data cannot be recovered — there is no CTRL+Z

4.3 Safety Tip — Preview Before Deleting

Best practice: Before running a DELETE, convert it to a SELECT to preview exactly which rows will be affected:

```
-- Step 1: Preview what will be deleted
SELECT * FROM EmployeeDemographics
WHERE EmployeeID = 1004

-- Step 2: Only if the result is correct, run the DELETE
DELETE FROM EmployeeDemographics
WHERE EmployeeID = 1004
```

- This lets you verify the right rows are targeted before committing the irreversible operation
- A WHERE mistake in a DELETE is one of the most costly errors in database management

SECTION 5: Joins — Combining Tables

5.1 What is a Join?

A JOIN combines rows from two or more tables into a single output, based on a matching condition — typically a shared column. The tables used in this course are joined on EmployeeID, which exists in both EmployeeDemographics and EmployeeSalary.

- Use joins when the data you need is spread across multiple tables
- Always specify which table each column comes from when column names exist in multiple tables: TableName.ColumnName
- The ON clause defines the matching condition between the two tables

5.2 Types of Joins

Join Type	What It Returns
INNER JOIN	Only rows where there is a match in BOTH tables. Rows with no match in either table are excluded.
LEFT (OUTER) JOIN	ALL rows from the LEFT (first) table, plus matching rows from the right table. Non-matching rows from the right table are shown as NULL.
RIGHT (OUTER) JOIN	ALL rows from the RIGHT (second) table, plus matching rows from the left table. Non-matching rows from the left table are shown as NULL.
FULL OUTER JOIN	ALL rows from BOTH tables. Rows with no match in either direction are shown with NULLs in the non-matching side.

5.3 Join Examples

INNER JOIN — employees with both demographics and salary on record:

```
SELECT dem.EmployeeID, dem.FirstName, dem.LastName, sal.Salary
FROM EmployeeDemographics dem
INNER JOIN EmployeeSalary sal
  ON dem.EmployeeID = sal.EmployeeID
```

LEFT JOIN — all employees from demographics, with salary if available (NULL if not):

```
SELECT dem.EmployeeID, dem.FirstName, sal.Salary
FROM EmployeeDemographics dem
LEFT JOIN EmployeeSalary sal
  ON dem.EmployeeID = sal.EmployeeID
```

5.4 Real-World Join Use Cases

Use case 1 — Finding the highest-paid employee (for Michael Scott's quarterly quota problem):

```
SELECT dem.EmployeeID, dem.FirstName, dem.LastName, sal.Salary
FROM EmployeeDemographics dem
INNER JOIN EmployeeSalary sal ON dem.EmployeeID = sal.EmployeeID
WHERE dem.FirstName <> 'Michael'
ORDER BY sal.Salary DESC
```

- Result: Dwight Schrute is the highest paid — he will have his pay cut

Use case 2 — Angela calculating the average salary for Salesmen:

```
SELECT sal.JobTitle, AVG(sal.Salary) AS AvgSalary
FROM EmployeeDemographics dem
INNER JOIN EmployeeSalary sal ON dem.EmployeeID = sal.EmployeeID
WHERE sal.JobTitle = 'Salesman'
GROUP BY sal.JobTitle
```

- Result: Salesmen earn an average of $52,000

5.5 Specifying Columns When Joining

- When a column name exists in both tables (e.g. EmployeeID) you MUST prefix it with the table name or alias to avoid an 'ambiguous column' error
- Use aliases to keep code readable:

```
FROM EmployeeDemographics dem
  -- 'dem' is the alias for EmployeeDemographics
FROM EmployeeSalary sal
  -- 'sal' is the alias for EmployeeSalary
```

SECTION 6: UNION and UNION ALL — Stacking Tables

6.1 UNION vs JOIN

Feature	JOIN
What it does	Combines tables SIDE BY SIDE (adds columns) based on a matching key
Requires matching column?	Yes — joined ON a common column
Duplicate handling	Does not deduplicate
Column names in output	From first/specified table

6.2 UNION — Removes Duplicates

UNION combines two SELECT statements and removes duplicate rows from the combined result (like a DISTINCT applied across both sets):

```
SELECT EmployeeID, FirstName, LastName, Age, Gender
FROM EmployeeDemographics
UNION
SELECT EmployeeID, FirstName, LastName, Age, Gender
FROM WarehouseEmployeeDemographics
```

- If an employee (e.g. Daryl Philbin) appears in both tables with identical data, UNION shows them only once
- Both SELECT statements must return the same number of columns with compatible data types

6.3 UNION ALL — Keeps All Rows

UNION ALL combines the result sets and keeps every row including duplicates:

```
SELECT EmployeeID, FirstName, Age FROM EmployeeDemographics
UNION ALL
SELECT EmployeeID, FirstName, Age FROM WarehouseEmployeeDemographics
ORDER BY EmployeeID
```

- Use UNION ALL when you intentionally want to see all records including duplicates — it is also faster than UNION since no deduplication step is needed

6.4 UNION with Different Tables — Data Type Warning

You can UNION tables with different structures as long as the number of columns and data types match. However mixing incompatible columns produces misleading results:

```
SELECT EmployeeID, FirstName, Age FROM EmployeeDemographics
UNION
SELECT EmployeeID, JobTitle, Salary FROM EmployeeSalary
```

- This technically works (INT matches INT, VARCHAR matches VARCHAR) but Age and Salary appear in the same column — a Salary of 45000 appears as if it were an Age
- Rule: when UNIONing different tables, make sure the columns you are selecting are logically equivalent — not just structurally compatible

SECTION 7: Aliasing — Renaming for Readability

7.1 What is Aliasing?

Aliasing is the practice of temporarily renaming a column or table within a query. Aliases exist only for the duration of that query — they do not change the actual database structure. Their primary purpose is to make queries more readable and maintainable.

- Column alias: renames a column in the output
- Table alias: creates a shorthand name for a table — essential when joining multiple tables
- AS keyword is optional — a space between the original name and the alias also works

7.2 Column Aliasing

Rename derived or aggregate columns so they have a meaningful header:

```
SELECT AVG(Age) AS AverageAge FROM EmployeeDemographics
```

- Without AS, an aggregate column is labelled '(No column name)' — always alias aggregates

Concatenating columns into one — alias makes the combined column meaningful:

```
SELECT FirstName + ' ' + LastName AS FullName
FROM EmployeeDemographics
```

- Combines FirstName and LastName with a space into a single column called FullName
- Without the alias, the concatenated column would have no name in the output

7.3 Table Aliasing

Table aliases are especially important when joining multiple tables — they avoid repeatedly typing the full table name:

```
SELECT dem.EmployeeID, dem.FirstName, sal.Salary
FROM EmployeeDemographics dem
INNER JOIN EmployeeSalary sal
  ON dem.EmployeeID = sal.EmployeeID
```

- dem is the alias for EmployeeDemographics; sal is the alias for EmployeeSalary
- Once you alias a table, you MUST use the alias (not the full name) when referencing that table's columns in the same query

7.4 Aliasing Best Practices

- Use meaningful aliases — demo, sal, wh (for warehouse) rather than a, b, c
- Single-letter aliases (a, b, c) are strongly discouraged — they make complex queries very hard to read
- Always alias aggregate function results — never leave a derived column with no name
- Consistent aliasing is essential when handing queries off to teammates or revisiting them months later

Example of BAD aliasing (avoid):

```
SELECT a.EmployeeID, b.Salary, c.Department
FROM EmployeeDemographics a
```

```
JOIN EmployeeSalary b ON a.EmployeeID = b.EmployeeID
JOIN EmployeeDepartment c ON a.EmployeeID = c.EmployeeID
```

Example of GOOD aliasing:

```
SELECT dem.EmployeeID, sal.Salary, dept.Department
FROM EmployeeDemographics dem
JOIN EmployeeSalary sal ON dem.EmployeeID = sal.EmployeeID
JOIN EmployeeDepartment dept ON dem.EmployeeID = dept.EmployeeID
```

SECTION 8: PARTITION BY — Window Functions

8.1 PARTITION BY vs GROUP BY

PARTITION BY is used inside a window function (OVER clause). It is often confused with GROUP BY because both involve grouping data. The critical difference is in what they do to the result set:

Feature	GROUP BY
Effect on rows	Reduces rows — rolls up all rows in a group into a single summary row
Used with	Aggregate functions in the SELECT clause
Can include non-grouped columns?	No — all non-aggregate SELECT columns must be in GROUP BY
Typical output	One row per group

8.2 PARTITION BY Syntax and Example

```
SELECT
  dem.FirstName,
  dem.LastName,
  dem.Gender,
  sal.Salary,
  COUNT(dem.Gender) OVER (PARTITION BY dem.Gender) AS TotalGender
FROM EmployeeDemographics dem
JOIN EmployeeSalary sal ON dem.EmployeeID = sal.EmployeeID
```

- Result: every individual row is returned, and the TotalGender column shows how many total employees share the same gender — e.g. Pam Beasley, Female, $36,000, 3 (3 total females)
- A GROUP BY version can only return Gender and its count — it cannot simultaneously show individual names and salaries
- Window function syntax: AGGREGATE(column) OVER (PARTITION BY column)
- OVER() with no PARTITION BY applies the aggregate across all rows (equivalent to a total):

```
AVG(Salary) OVER () AS CompanyAvgSalary
```

SECTION 9: CTEs and Temp Tables — Reusable Result Sets

9.1 CTE — Common Table Expression

A CTE (Common Table Expression) is a named temporary result set that exists only for the duration of the single query it precedes. It is defined using the WITH keyword and acts like a named subquery that you can reference in the main SELECT below it.

```
WITH CTE_Employee AS
(
  SELECT
    dem.FirstName,
    dem.LastName,
    dem.Gender,
    sal.Salary,
    COUNT(dem.Gender) OVER (PARTITION BY dem.Gender) AS TotalGender,
    AVG(sal.Salary) OVER (PARTITION BY dem.Gender) AS AvgSalary
  FROM EmployeeDemographics dem
  JOIN EmployeeSalary sal ON dem.EmployeeID = sal.EmployeeID
)
SELECT FirstName, AvgSalary
FROM CTE_Employee
```

- The WITH block defines the CTE; the SELECT below it queries the CTE
- The SELECT must immediately follow the closing parenthesis of the CTE — you cannot write other statements between them
- CTEs are sometimes called 'WITH queries'
- CTEs are NOT stored anywhere — running only the SELECT FROM CTE_Employee without the WITH block produces an error
- Each time the full query runs, SQL recreates the CTE fresh

9.2 Temp Tables

A Temporary Table is a table created in SQL Server's tempdb system database. It works just like a regular table but is automatically dropped when the session ends. Unlike a CTE, a temp table can be queried multiple times within the same session.

```
CREATE TABLE #TempEmployee
(
  JobTitle     VARCHAR(50),
  EmployeesPerJob INT,
  AvgAge       INT,
  AvgSalary    INT
)

INSERT INTO #TempEmployee
SELECT
  sal.JobTitle,
  COUNT(sal.JobTitle),
  AVG(dem.Age),
  AVG(sal.Salary)
FROM EmployeeDemographics dem
JOIN EmployeeSalary sal ON dem.EmployeeID = sal.EmployeeID
```

```
GROUP BY sal.JobTitle

SELECT * FROM #TempEmployee
```

- The # prefix makes it a temp table — this is the only difference from a regular CREATE TABLE
- You can also INSERT into a temp table by selecting from an existing table — no need to manually type values
- Use DROP TABLE IF EXISTS to prevent errors when rerunning:
  ```
  DROP TABLE IF EXISTS #TempEmployee
  ```
- This is especially important in stored procedures that may run multiple times

9.3 CTE vs Temp Table vs Subquery — Comparison

Feature	Subquery
Storage	Not stored — runs inline
Reusability	Must be rewritten every time
Performance	Slowest for complex logic
Readability	Can become hard to read when nested
Best for	Quick, one-off inline filtering

SECTION 10: String Functions — Cleaning and Transforming Text

10.1 Why String Functions Matter

In real-world data, text data is often messy — extra spaces, incorrect capitalisation, unwanted characters, and name inconsistencies are extremely common. SQL string functions allow you to clean and transform this data at query time without modifying the underlying table.

10.2 TRIM, LTRIM, RTRIM — Removing Whitespace

Function	What It Removes
TRIM(column)	Removes spaces from BOTH the left and right sides
LTRIM(column)	Removes spaces from the LEFT side only
RTRIM(column)	Removes spaces from the RIGHT side only

```
SELECT
  EmployeeID,
  TRIM(EmployeeID) AS Trimmed,
  LTRIM(EmployeeID) AS LeftTrimmed,
  RTRIM(EmployeeID) AS RightTrimmed
FROM EmployeeErrors
```

- Extra whitespace is invisible in the output — TRIM is essential when importing data from external sources like CSV files or spreadsheets

10.3 REPLACE — Substituting Substrings

REPLACE finds all occurrences of a substring and replaces them with a new value:

```
SELECT
  LastName,
  REPLACE(LastName, '-Fired', '') AS LastName_Fixed
FROM EmployeeErrors
```

- Replaces the string '-Fired' with an empty string (effectively removing it)
- REPLACE(column, find, replace_with) — all three arguments are required
- Case-sensitive by default on some SQL Server collations

10.4 SUBSTRING — Extracting Part of a String

SUBSTRING extracts a portion of a string starting at a specified position and going forward a specified number of characters:

```
SUBSTRING(column, start_position, length)

SELECT FirstName, SUBSTRING(FirstName, 1, 3) AS First3
FROM EmployeeErrors
```

- SUBSTRING(FirstName, 1, 3) — start at character 1, return 3 characters → 'Jim' from 'Jimbo', 'Pam' from 'Pamela'
- SUBSTRING(FirstName, 3, 3) — start at character 3, return 3 characters → 'mbo' from 'Jimbo'

10.5 Fuzzy Matching with SUBSTRING

Fuzzy matching uses SUBSTRING (or similar) to match records across tables when the values are slightly different — for example 'Alex' in one table and 'Alexander' in another. Joining on full name would fail; joining on the first 4 characters can succeed:

```
SELECT err.FirstName, dem.FirstName
FROM EmployeeErrors err
JOIN EmployeeDemographics dem
  ON SUBSTRING(err.FirstName, 1, 3) = SUBSTRING(dem.FirstName, 1, 3)
```

- 'Jimbo' and 'Jim' both produce 'Jim' from SUBSTRING(name, 1, 3) — they match
- 'Pamela' and 'Pam' both produce 'Pam' — they match
- In practice, fuzzy match on multiple fields for accuracy: first name + last name + gender + age + date of birth
- The more fields that match, the higher the confidence that two records refer to the same person

10.6 UPPER and LOWER — Case Conversion

```
SELECT FirstName, LOWER(FirstName) AS LowerName, UPPER(FirstName) AS UpperName
FROM EmployeeErrors
```

- LOWER('TOBY') → 'toby' — converts all characters to lowercase
- UPPER('toby') → 'TOBY' — converts all characters to uppercase
- Useful for normalising data before comparisons — 'Jim' = 'JIM' fails by default in case-sensitive comparisons

10.7 String Functions Quick Reference

Function	Syntax
TRIM	TRIM(column)
LTRIM	LTRIM(column)
RTRIM	RTRIM(column)
REPLACE	REPLACE(column, find, new)
SUBSTRING	SUBSTRING(column, start, length)
UPPER	UPPER(column)
LOWER	LOWER(column)
LEN	LEN(column)
LEFT	LEFT(column, n)
RIGHT	RIGHT(column, n)

SECTION 11: Stored Procedures and Subqueries

11.1 Stored Procedures

A stored procedure is a saved, named group of SQL statements stored permanently in the database. Think of it as a reusable function for SQL. You write it once and call it by name whenever you need it.

- Benefits:
  - Reusability — one stored procedure can be executed by multiple users and applications
  - Performance — SQL Server compiles and caches the execution plan, reducing overhead
  - Reduced network traffic — send one EXEC command instead of the full query text
  - Maintainability — update the procedure once and all callers automatically get the update
  - Security — users can be given permission to run a stored procedure without being granted direct table access

Creating a simple stored procedure:

```
CREATE PROCEDURE Test
AS
SELECT * FROM EmployeeDemographics

-- Executing it:
EXEC Test
```

Creating a stored procedure with a parameter:

```
ALTER PROCEDURE TempEmployee
  @JobTitle NVARCHAR(100)
AS
SELECT *
FROM #TempEmployee
WHERE JobTitle = @JobTitle

-- Executing with a parameter:
EXEC TempEmployee @JobTitle = 'Salesman'
```

- Parameters allow one stored procedure to return different results based on the input — like a function argument
- You can have multiple parameters — just add more @ParameterName DATATYPE declarations
- Use ALTER PROCEDURE to modify an existing stored procedure
- After creating or modifying a stored procedure, refresh the Stored Procedures folder in SSMS to see it

11.2 Subqueries — Queries Inside Queries

A subquery is a SELECT statement nested inside another SQL statement. Subqueries can appear in the SELECT, FROM, or WHERE clause. They allow you to use the result of one query as input to another.

Subquery in SELECT — computing a comparison value alongside each row:

```
SELECT
  EmployeeID,
  Salary,
  (SELECT AVG(Salary) FROM EmployeeSalary) AS AllAvgSalary
FROM EmployeeSalary
```

- Returns each employee's salary alongside the company average — similar to using PARTITION BY without grouping
- The inner (SELECT AVG(Salary) ...) runs once and its result is placed in every row of the outer query

Subquery in FROM — treating a query result as a temporary table:

```
SELECT a.EmployeeID, a.AllAvgSalary
FROM (
  SELECT
    EmployeeID,
    Salary,
    COUNT(Gender) OVER (PARTITION BY Gender) AS GenderCount,
    AVG(Salary) OVER (PARTITION BY Gender) AS AllAvgSalary
  FROM EmployeeDemographics dem
  JOIN EmployeeSalary sal ON dem.EmployeeID = sal.EmployeeID
) a
```

- The inner query produces a derived table; the outer query selects from it
- This is the equivalent of what a CTE or temp table does — but written inline
- Less readable and less reusable than a CTE or temp table; generally not recommended for complex logic

Subquery in WHERE — filtering based on another table's data:

```
SELECT EmployeeID, JobTitle, Salary
FROM EmployeeSalary
WHERE EmployeeID IN
(
  SELECT EmployeeID
  FROM EmployeeDemographics
  WHERE Age > 30
)
```

- Returns salary records only for employees who are over 30 — Age comes from a different table
- The subquery in WHERE must return exactly ONE column — not SELECT * — only the column being matched
- IN allows matching against a list of values returned by the subquery
- Alternative: JOIN the two tables on EmployeeID and add WHERE dem.Age > 30 — both approaches work

11.3 When to Use Each Approach

Approach	Best Used When
Subquery in SELECT	You need a scalar value (one result) displayed alongside each row — e.g. company average salary next to each employee's salary
Subquery in FROM	You need to transform data before querying it — but prefer CTE or temp table for readability
Subquery in WHERE	You need to filter rows in one table based on criteria from a different table and don't want to write a full JOIN
CTE	You want clean, readable code for a complex multi-step transformation used once per query
Temp Table	You need to compute data once and reference it multiple times, or you are building a stored procedure
JOIN	Two tables share a key and you want to combine their columns into one result set

SECTION 12: SQL Quick Reference

12.1 Core SQL Syntax Cheat Sheet

Operation	Syntax / Example
Select all	SELECT * FROM TableName
Select columns	SELECT Col1, Col2 FROM TableName
Top N rows	SELECT TOP 10 * FROM TableName
Distinct values	SELECT DISTINCT Col FROM TableName
Column alias	SELECT AVG(Salary) AS AvgSalary FROM Table
Filter rows	SELECT * FROM Table WHERE Age > 30
Multiple conditions	WHERE Age > 30 AND Gender = 'Male'
Pattern match	WHERE Name LIKE 'J%'
List match	WHERE JobTitle IN ('Manager', 'Accountant')
Null check	WHERE Age IS NULL
Aggregate count	SELECT COUNT(*) FROM Table
Group and count	SELECT Gender, COUNT(*) FROM Table GROUP BY Gender
Filter groups	GROUP BY JobTitle HAVING COUNT(*) > 1
Sort ascending	ORDER BY Salary ASC
Sort descending	ORDER BY Salary DESC
Insert row	INSERT INTO Table VALUES (101, 'Jim', 'Halpert', 30, 'Male')
Update row	UPDATE Table SET Age = 31 WHERE EmployeeID = 102
Delete row	DELETE FROM Table WHERE EmployeeID = 105
Inner join	FROM TableA INNER JOIN TableB ON TableA.ID = TableB.ID
Left join	FROM TableA LEFT JOIN TableB ON TableA.ID = TableB.ID
Union	SELECT ... FROM A UNION SELECT ... FROM B
Union all	SELECT ... FROM A UNION ALL SELECT ... FROM B
CTE	WITH CTE_Name AS (SELECT ...) SELECT ... FROM CTE_Name
Temp table	CREATE TABLE #Temp (...) -- prefix with #
Drop temp table	DROP TABLE IF EXISTS #Temp
String trim	TRIM(column) / LTRIM / RTRIM
String replace	REPLACE(column, 'old', 'new')
Substring	SUBSTRING(column, start, length)
Uppercase	UPPER(column) / LOWER(column)
Partition by	COUNT(*) OVER (PARTITION BY Gender)
Create procedure	CREATE PROCEDURE Name AS SELECT ...
Execute procedure	EXEC ProcedureName @Param = 'Value'
Subquery in WHERE	WHERE ID IN (SELECT ID FROM Table2 WHERE Age > 30)