Structured Query Language

UNIT III

DATABASE MANAGEMENT SYSTEM

A comprehensive guide to SQL fundamentals, data manipulation, and database integrity for ICT diploma engineering students

COURSE CONTEXT

Understanding SQL in Database Management

Structured Query Language (SQL) is the universal language for managing and manipulating relational databases. As a fundamental skill for ICT professionals, SQL enables you to create, modify, and retrieve data efficiently from database systems.

This unit covers everything from basic data types to advanced integrity constraints, providing you with the essential tools to design and manage robust database applications. You'll learn how to write queries that extract meaningful information and implement safeguards that protect data quality.

Learning Objectives

- Master SQL syntax and commands
- Manipulate data effectively
- Apply integrity constraints
- Write complex queries

CHAPTER OVERVIEW

What We'll Cover

01

SQL Fundamentals

Data types, DDL commands, and database structure creation

02

Data Manipulation

DML commands for inserting, updating, and querying data

03

Functions & Operators

Built-in functions for data processing and calculations

04

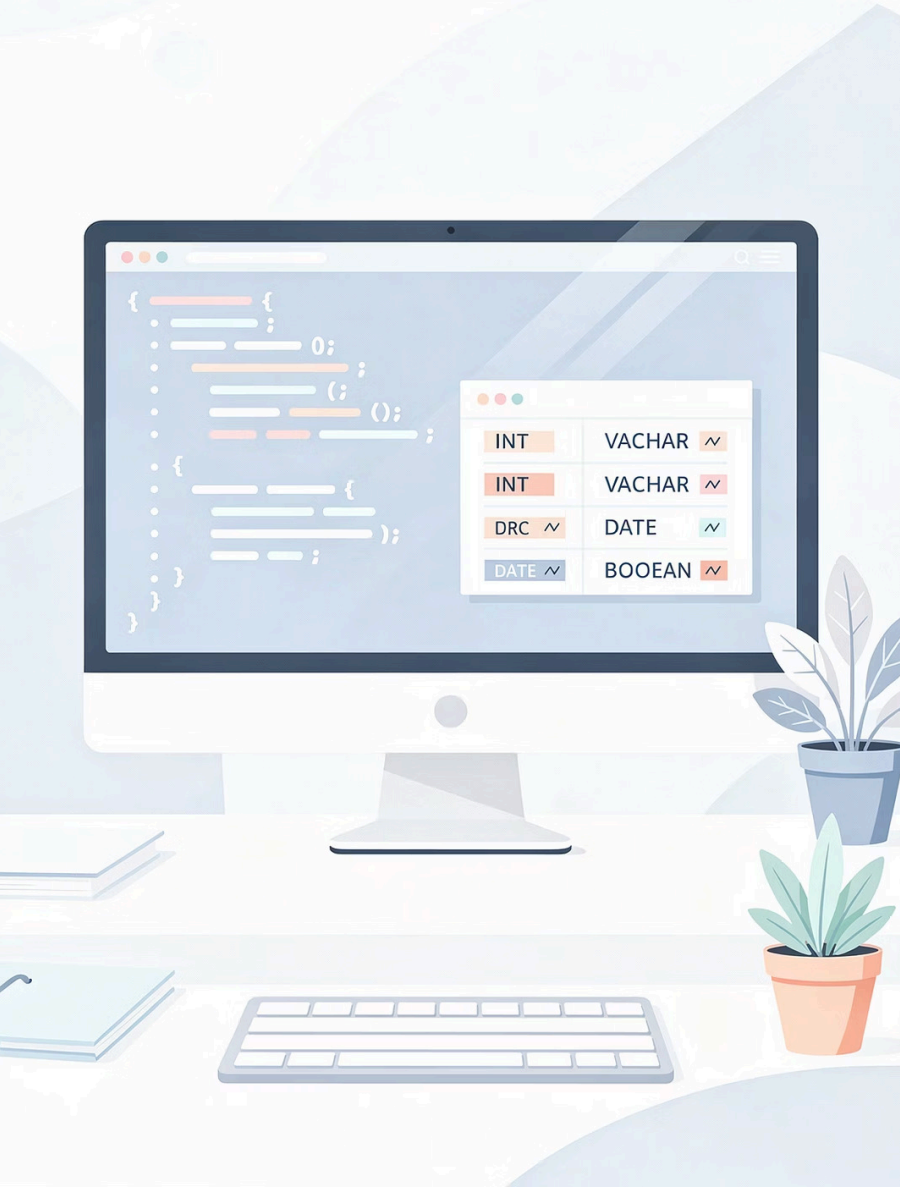
Advanced Queries

Joins, set operators, and complex query techniques

05

Data Integrity

Constraints that ensure database reliability and consistency



3.1

SQL Data Types

Understanding data types is foundational to database design. SQL data types define what kind of information can be stored in each column, ensuring data consistency and optimizing storage.

Common SQL Data Types

Character Types

- **CHAR(n)**: Fixed-length strings
- **VARCHAR(n)**: Variable-length strings
- **TEXT**: Large text data

Use VARCHAR for variable-length data like names and addresses to save storage space.

Numeric Types

- **INT**: Whole numbers
- **DECIMAL(p,s)**: Fixed-point numbers
- **FLOAT**: Floating-point numbers

Choose DECIMAL for financial calculations where precision is critical.

Date & Time

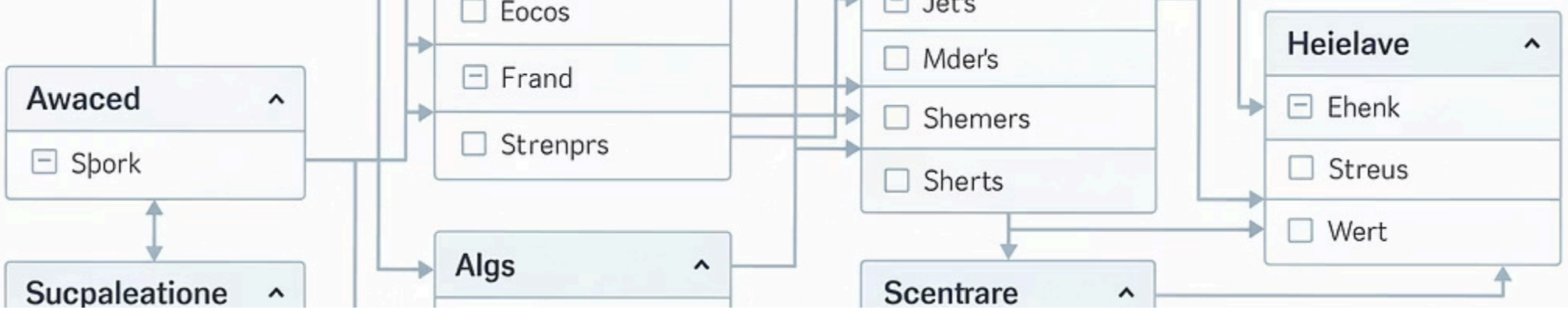
- **DATE**: Calendar dates
- **TIME**: Time of day
- **DATETIME**: Date and time combined

DATETIME is ideal for tracking events with both date and time components.

Boolean & Binary

- **BOOLEAN**: True/false values
- **BLOB**: Binary large objects
- **BINARY**: Fixed binary data

Use BLOB for storing images, documents, and other binary files.



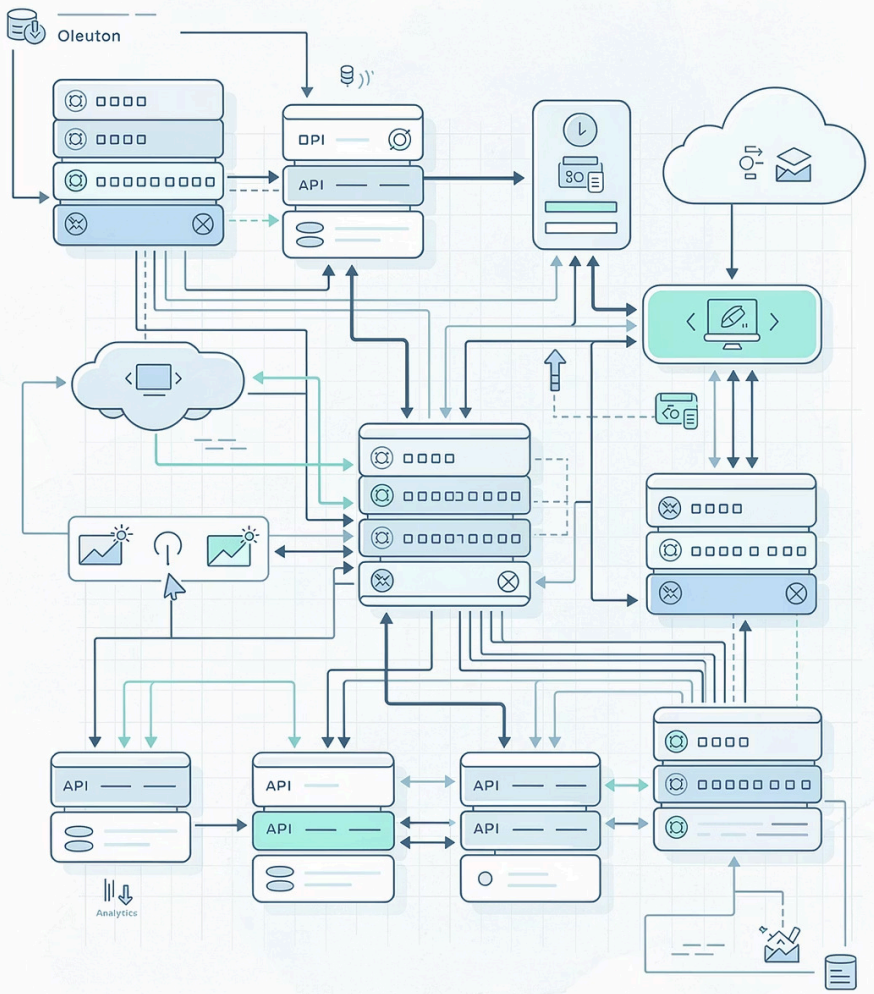
Choosing the Right Data Type

Best Practices

- Select the smallest data type that accommodates your data range to optimize storage
- Use VARCHAR instead of CHAR for variable-length text to save space
- Choose DECIMAL over FLOAT for monetary values to avoid rounding errors
- Consider future data growth when setting size limits

Common Mistakes to Avoid

- Using oversized data types that waste storage
- Storing dates as strings, losing built-in date functions
- Using FLOAT for currency calculations
- Not specifying length for VARCHAR columns



3.2

Data Definition Language (DDL) Commands

DDL commands are the foundation of database structure. They allow you to create, modify, and delete database objects like tables, indexes, and schemas. Understanding DDL is essential for designing efficient database architectures.

The CREATE Command

The CREATE command builds new database objects. When creating tables, you define column names, data types, and constraints that govern data quality.

Basic Syntax

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype
);
```

Practical Example

```
CREATE TABLE Students (
    StudentID INT,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    EnrollmentDate DATE
);
```

Key Considerations

- Choose meaningful table and column names
- Plan your data types carefully
- Consider future expansion needs
- Document your schema design

The CREATE command can also build indexes, views, and other database objects that enhance performance and functionality.

The ALTER Command

ALTER modifies existing database structures without losing data. This command is crucial for adapting databases to evolving business requirements.

Add New Column

```
ALTER TABLE Students
ADD Email VARCHAR(100);
```

Extends table structure with additional fields as requirements change.

Modify Column

```
ALTER TABLE Students
MODIFY Email VARCHAR(150);
```

Changes data type or size of existing columns to accommodate new needs.

Drop Column

```
ALTER TABLE Students
DROP COLUMN Email;
```

Removes unnecessary columns, though this permanently deletes the data.

TRUNCATE vs DROP Commands

TRUNCATE

TRUNCATE removes all rows from a table but keeps the table structure intact. It's faster than DELETE because it doesn't generate individual row delete operations.

```
TRUNCATE TABLE Students;
```

- Removes all data quickly
- Preserves table structure
- Cannot be rolled back in some databases
- Resets auto-increment counters

DROP

DROP completely removes a table and its structure from the database. This is irreversible and should be used with extreme caution in production environments.

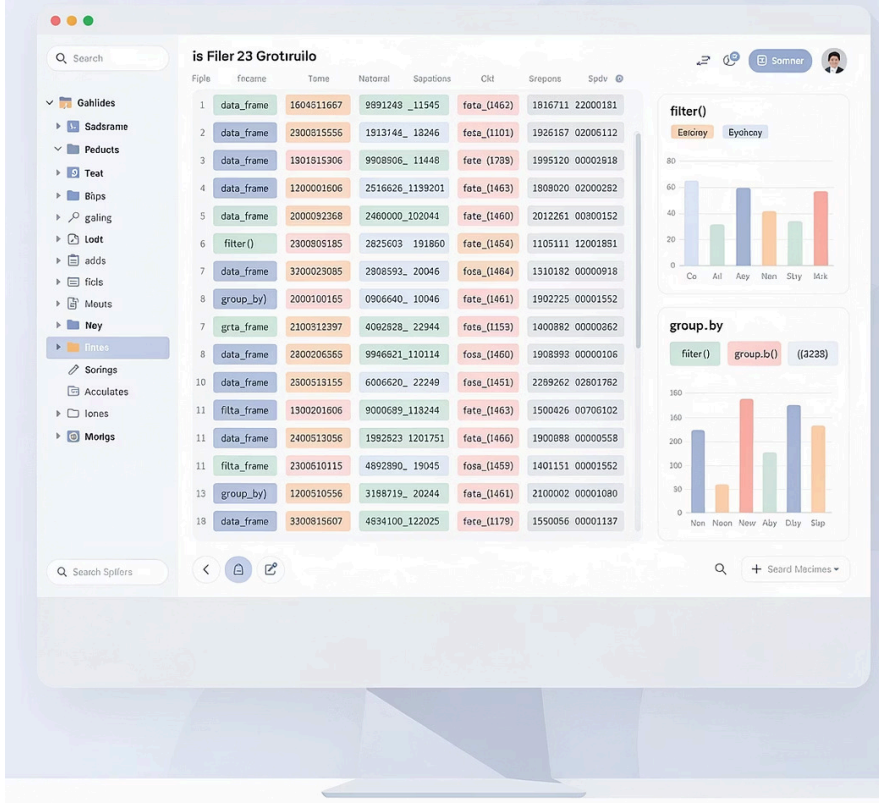
```
DROP TABLE Students;
```

- Deletes entire table permanently
- Removes all data and structure
- Cannot be undone
- Frees up storage space

3.3

Data Manipulation Language (DML) Commands

DML commands are the workhorses of database operations. They enable you to insert new data, retrieve information, update existing records, and remove obsolete data. Mastering DML is essential for effective database management.



The INSERT Command

INSERT adds new records to your tables. Proper use of INSERT ensures data is entered consistently and completely.

1

Specify All Columns

```
INSERT INTO Students
VALUES (1, 'John',
'Doe', '2024-01-15');
```

2

Specify Selected Columns

```
INSERT INTO Students
(StudentID, FirstName)
VALUES (2, 'Jane');
```

3

Insert Multiple Rows

```
INSERT INTO Students
VALUES (3, 'Bob',
'Smith', '2024-01-16'),
(4, 'Alice', 'Johnson',
'2024-01-17');
```

Always validate data before insertion to maintain database integrity. Consider using transactions for multiple related inserts to ensure all-or-nothing execution.

The SELECT Command

SELECT is the most frequently used SQL command, allowing you to retrieve and view data from your tables. It forms the basis for data analysis and reporting.

Basic Selection

```
SELECT * FROM Students;
```

Retrieves all columns and rows from the Students table.

Specific Columns

```
SELECT FirstName, LastName
FROM Students;
```

Returns only the specified columns, improving query performance.

Filtered Results

```
SELECT * FROM Students
WHERE EnrollmentDate >
'2024-01-01';
```

Uses WHERE clause to filter results based on conditions.

Sorted Output

```
SELECT * FROM Students
ORDER BY LastName ASC;
```

Arranges results in ascending or descending order.

UPDATE and DELETE Commands

UPDATE: Modifying Existing Data

UPDATE changes values in existing records. Always use a WHERE clause to avoid accidentally updating all rows.

```
UPDATE Students
SET Email = 'john.doe@example.com'
WHERE StudentID = 1;
```

- Modifies specific column values
- Requires WHERE clause for targeted updates
- Can update multiple columns simultaneously

DELETE: Removing Records

DELETE removes specific rows from a table. Use with caution as deleted data cannot be easily recovered without backups.

```
DELETE FROM Students
WHERE StudentID = 1;
```

- Permanently removes rows
- WHERE clause prevents accidental mass deletion
- Can be rolled back within a transaction

3.4

Privilege Commands: GRANT and REVOKE

Database security relies on controlling who can access and modify data. Privilege commands manage user permissions, ensuring that only authorized individuals can perform specific operations on database objects.

The GRANT Command

GRANT assigns specific privileges to users or roles, enabling controlled access to database resources.

```
GRANT SELECT, INSERT
ON Students
TO user_name;
```

- Assigns read and write permissions
- Can be applied to specific tables or entire databases
- Supports multiple privilege types

The REVOKE Command

REVOKE removes previously granted privileges, immediately restricting user access to specified operations.

```
REVOKE INSERT
ON Students
FROM user_name;
```

- Removes specific permissions
- Takes effect immediately
- Essential for maintaining security

Common Privilege Types

SELECT

Allows reading data from tables. This is the most basic privilege needed for viewing information without modification capabilities.

INSERT

Permits adding new records to tables. Essential for users who need to create new entries in the database.

UPDATE

Enables modifying existing records. Grant carefully as this can alter critical data if misused.

DELETE

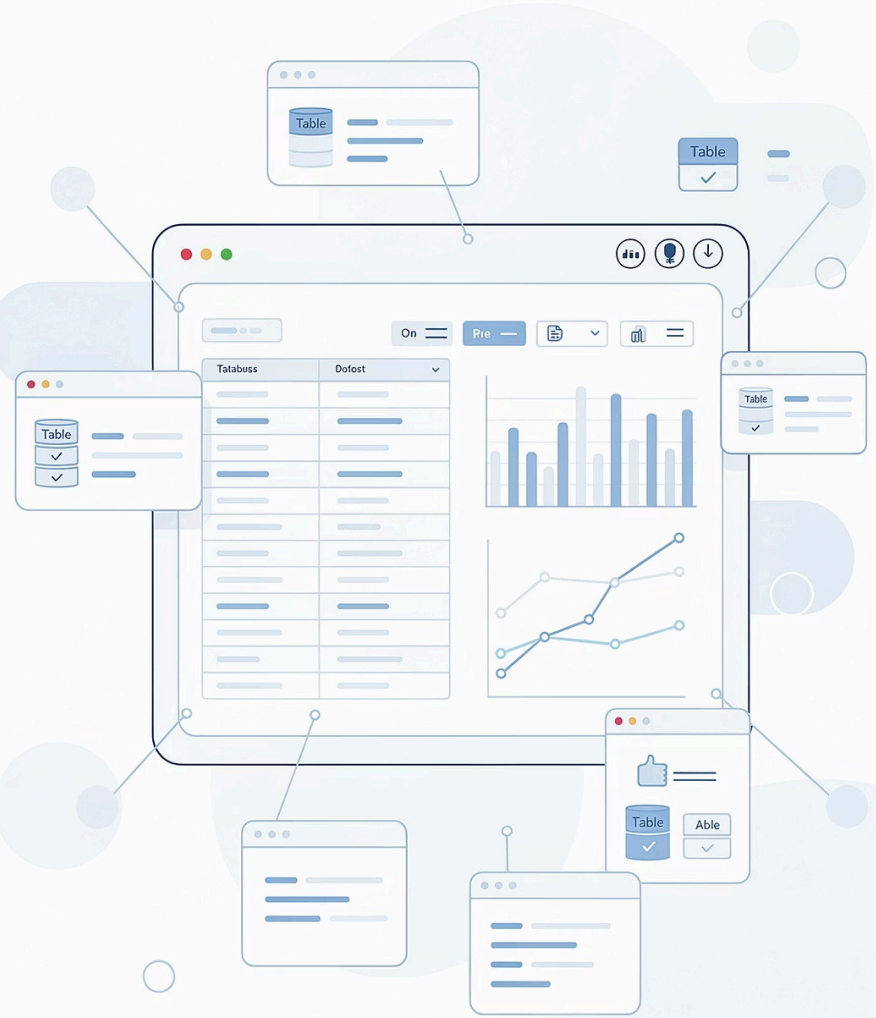
Allows removal of records. This is a powerful privilege that should be restricted to trusted users.

ALTER

Permits modification of table structure. Typically reserved for database administrators and developers.

ALL PRIVILEGES

Grants complete control over database objects. Use sparingly and only for administrative accounts.



3.5

SQL Views

Views are virtual tables created by stored queries. They simplify complex queries, enhance security by limiting data exposure, and provide a consistent interface to underlying table structures that may change over time.

Creating and Using Views

What Are Views?

A view is a saved SELECT query that appears as a table. Unlike real tables, views don't store data—they dynamically retrieve it from underlying tables when queried.

Creating a View

```
CREATE VIEW ActiveStudents AS
SELECT StudentID, FirstName,
LastName
FROM Students
WHERE Status = 'Active';
```

Advantages of Views

- **Simplified Queries:** Hide complex joins and calculations behind simple table-like interfaces
- **Security:** Restrict access to specific columns or rows without exposing entire tables
- **Consistency:** Ensure users always see data in a standardized format
- **Abstraction:** Isolate applications from underlying schema changes

Using a View

```
SELECT * FROM ActiveStudents;
```

Query views exactly like regular tables.

View Best Practices

Name Views Clearly

Use descriptive names that indicate the view's purpose, such as "vw_ActiveStudents" or "SalesReport_Q1". Consistent naming conventions improve maintainability.

Document View Logic

Complex views should include comments explaining their purpose and the business logic they implement. This helps future developers understand and maintain them.

Optimize Performance

Views that join many tables or perform complex calculations can slow queries. Consider indexed views or materialized views for frequently accessed data.

Limit Updateable Views

While some views allow INSERT, UPDATE, and DELETE operations, complex views involving joins or aggregations typically cannot be updated. Design accordingly.

FUNCTIONS

SQL Functions

SQL provides a rich library of built-in functions that transform, calculate, and manipulate data. Understanding these functions is essential for writing efficient queries and performing complex data operations.

3.6

Single Row Functions

Single row functions operate on individual rows and return one result per row. They process data row-by-row, making them ideal for calculations and transformations within SELECT statements.

Characteristics

- Process one row at a time
- Return one result per input row
- Can be nested within each other
- Used in SELECT, WHERE, and ORDER BY clauses

Function Categories

- Character functions
- Numeric functions
- Date functions
- Conversion functions
- Miscellaneous functions

Common Uses

- Format output data
- Perform calculations
- Convert data types
- Handle null values
- Extract date components

3.7

Date Functions

Date functions manipulate temporal data, enabling you to calculate durations, extract components, and format dates for display or comparison.

1 CURRENT_DATE

Returns today's date

```
SELECT CURRENT_DATE;
```

2 DATE_ADD

Adds intervals to dates

```
DATE_ADD('2024-01-01',
INTERVAL 30 DAY)
```

3 DATEDIFF

Calculates date differences

```
DATEDIFF('2024-12-31',
'2024-01-01')
```

4 DATE_FORMAT

Formats date output

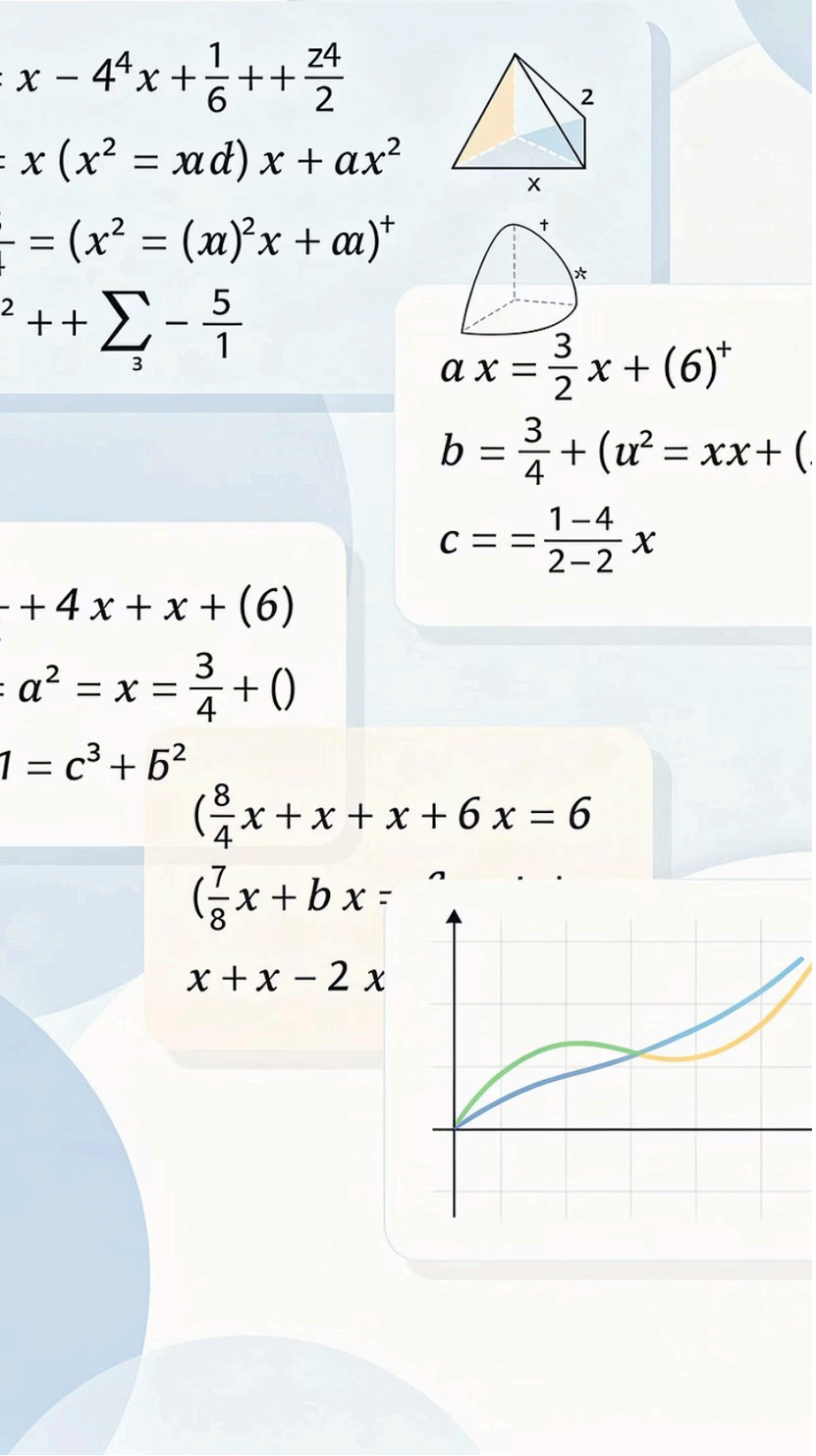
```
DATE_FORMAT(date,
'%Y-%m-%d')
```

Extracting Components

- **YEAR(date)**: Extracts the year
- **MONTH(date)**: Extracts the month number
- **DAY(date)**: Extracts the day of month
- **HOUR(time)**: Extracts the hour

Practical Applications

- Calculate age from birthdate
- Determine days until deadline
- Group records by month or year
- Filter records by date range



3.8

Numeric Functions

Numeric functions perform mathematical operations and calculations on numeric data types, from simple rounding to complex trigonometric calculations.

Rounding Functions

- **ROUND(n, d)**: Rounds to d decimal places
- **CEIL(n)**: Rounds up to nearest integer
- **FLOOR(n)**: Rounds down to nearest integer
- **TRUNC(n, d)**: Truncates to d decimals

```
SELECT ROUND(123.456, 2);
-- Returns: 123.46
```

Mathematical Operations

- **ABS(n)**: Absolute value
- **POWER(n, m)**: n raised to power m
- **SQRT(n)**: Square root
- **MOD(n, m)**: Remainder of n/m

```
SELECT POWER(2, 3);
-- Returns: 8
```

Statistical Functions

- **SIGN(n)**: Returns -1, 0, or 1
- **GREATEST(n1, n2, ...)**: Largest value
- **LEAST(n1, n2, ...)**: Smallest value

```
SELECT GREATEST(10, 25, 5);
-- Returns: 25
```

Character Functions

Character functions manipulate string data, allowing you to transform, search, and format text within your database queries.

Case Conversion

- **UPPER(str)**: Converts to uppercase
- **LOWER(str)**: Converts to lowercase
- **INITCAP(str)**: Capitalizes first letter of each word

```
SELECT UPPER('hello world');
-- Returns: HELLO WORLD
```

String Extraction

- **SUBSTR(str, start, length)**: Extracts substring
- **LEFT(str, n)**: First n characters
- **RIGHT(str, n)**: Last n characters

String Manipulation

- **CONCAT(str1, str2)**: Joins strings
- **TRIM(str)**: Removes leading/trailing spaces
- **REPLACE(str, old, new)**: Replaces text
- **LENGTH(str)**: Returns string length

```
SELECT CONCAT('Hello', ' ', 'World');
-- Returns: Hello World
```

Search Functions

- **INSTR(str, substr)**: Finds substring position
- **LIKE**: Pattern matching in WHERE clause

3.9

Conversion Functions

Conversion functions transform data from one type to another, essential for mixing different data types in queries and ensuring proper data handling.

1

TO_CHAR

Converts numbers or dates to character strings with formatting

```
TO_CHAR(12345.67,
'$99,999.99')
```

Result: $12,345.67

2

TO_NUMBER

Converts character strings to numeric values for calculations

```
TO_NUMBER('12345.67')
```

Result: 12345.67

3

TO_DATE

Converts strings to date data type using format masks

```
TO_DATE('2024-01-15',
'YYYY-MM-DD')
```

Result: 2024-01-15

Why Conversion Matters

Implicit conversion can lead to unexpected results or errors. Always use explicit conversion functions to ensure data is handled correctly, especially when comparing or calculating with mixed data types.

3.10

Miscellaneous Functions

These utility functions handle special cases like null values, conditional logic, and data validation, making your queries more robust and flexible.

NVL & COALESCE

Handle null values by providing default replacements

```
NVL(commission, 0)
COALESCE(phone, email,
'No contact')
```

NVL returns a specified value if the column is null. COALESCE returns the first non-null value from a list.

CASE Expression

Implements conditional logic within queries

```
CASE
  WHEN grade >= 90
    THEN 'A'
  WHEN grade >= 80
    THEN 'B'
  ELSE 'C'
END
```

Allows if-then-else logic directly in SQL statements.

NULLIF & NVL2

Advanced null handling for complex scenarios

```
NULLIF(value1, value2)
NVL2(expr, if_not_null,
if_null)
```

NULLIF returns null if two values are equal. NVL2 provides different results based on null status.

3.11

Group Functions

Group functions operate on sets of rows to produce a single summary value. These aggregate functions are fundamental to data analysis and reporting.

Essential Aggregate Functions

COUNT

Returns the number of rows in a group

```
SELECT COUNT(*)
FROM Students;
```

Use COUNT(column) to count non-null values, or COUNT(*) to count all rows.

SUM

Calculates the total of numeric values

```
SELECT SUM(Credits)
FROM Courses;
```

Adds all numeric values in a column, ignoring null values.

AVG

Computes the average of numeric values

```
SELECT AVG(Grade)
FROM Scores;
```

Calculates mean value, useful for analyzing central tendencies in data.

MAX

Finds the maximum value in a column

```
SELECT MAX(Salary)
FROM Employees;
```

Works with numeric, character, and date types to find highest value.

MIN

Identifies the minimum value in a column

```
SELECT MIN(Salary)
FROM Employees;
```

Finds lowest value across all rows in the specified column.

Using Group Functions Effectively

Important Rules

- Group functions ignore null values (except COUNT(*))
- Cannot mix group functions with individual columns without GROUP BY
- Can nest group functions to one level
- DISTINCT can be used within group functions

Example Query

```
SELECT Department,
    COUNT(*) as EmployeeCount,
    AVG(Salary) as AvgSalary,
    MAX(Salary) as MaxSalary
FROM Employees
GROUP BY Department;
```

This query demonstrates how group functions work with GROUP BY to produce summary statistics for each department, providing valuable insights into workforce distribution and compensation.

3.13-3.16

SQL Operators

Operators are symbols that tell SQL to perform specific mathematical, comparison, or logical operations. Understanding operators is crucial for constructing effective queries and filtering data precisely.

Arithmetic Operators

Arithmetic operators perform mathematical calculations on numeric data within SQL statements.

Addition (+)

Adds two numbers together

```
SELECT Price + Tax
FROM Products;
```

Subtraction (-)

Subtracts one number from another

```
SELECT Revenue - Cost
FROM Sales;
```

Multiplication (*)

Multiplies two numbers

```
SELECT Quantity * Price
FROM Orders;
```

Division (/)

Divides one number by another

```
SELECT Total / Count
FROM Summary;
```

Use parentheses to control order of operations. SQL follows standard mathematical precedence: multiplication and division before addition and subtraction.

Comparison Operators

Comparison operators evaluate relationships between values, returning true or false results used primarily in WHERE and HAVING clauses.

Equality Operators

- **=** Equal to
- **!=** or **<>** Not equal to

```
SELECT * FROM Students
WHERE Grade = 'A';

SELECT * FROM Products
WHERE Stock != 0;
```

Relational Operators

- **>** Greater than
- **<** Less than
- **>=** Greater than or equal
- **<=** Less than or equal

```
SELECT * FROM Employees
WHERE Salary > 50000;

SELECT * FROM Orders
WHERE OrderDate <=
'2024-01-31';
```

Special Operators

- **BETWEEN** Range check
- **IN** List membership
- **LIKE** Pattern matching
- **IS NULL** Null check

```
SELECT * FROM Products
WHERE Price BETWEEN
10 AND 100;

SELECT * FROM Students
WHERE Major IN ('CS',
'IT', 'SE');
```

Logical Operators

Logical operators combine multiple conditions, enabling complex filtering logic in SQL queries.

AND

Returns true only if all conditions are true

```
SELECT *
FROM Students
WHERE Grade = 'A'
  AND Attendance >
90;
```

Both conditions must be satisfied for a row to be included in results.

OR

Returns true if any condition is true

```
SELECT *
FROM Employees
WHERE Dept =
'Sales'
  OR Dept =
'Marketing';
```

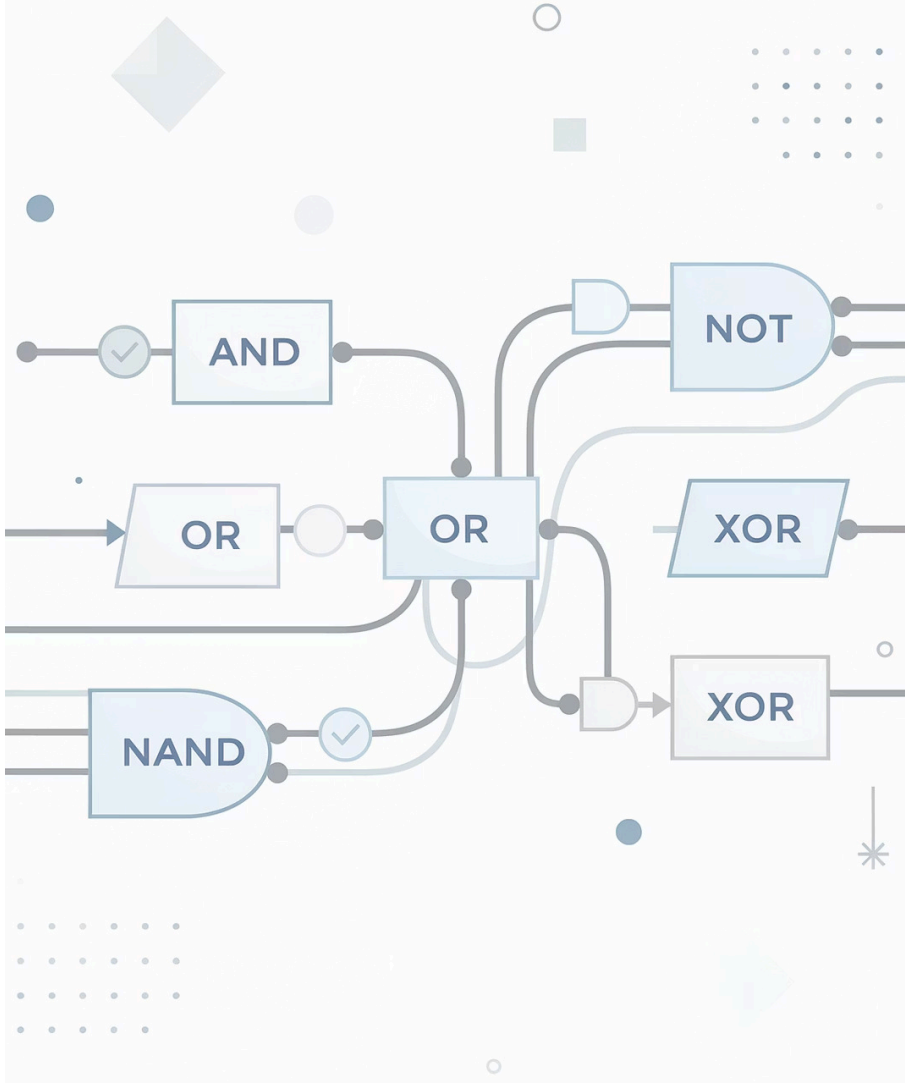
Rows matching either condition are included in the result set.

NOT

Negates a condition

```
SELECT *
FROM Products
WHERE NOT
  Category =
'Electronics';
```

Returns all rows that don't match the specified condition.



3.17

GROUP BY, HAVING, and ORDER BY Clauses

These clauses organize and filter query results, enabling sophisticated data analysis and presentation. They work together to create powerful analytical queries.

The GROUP BY Clause

GROUP BY divides rows into groups based on column values, allowing aggregate functions to operate on each group separately rather than the entire table.

Basic Syntax

```
SELECT Department, COUNT(*),
       AVG(Salary)
FROM Employees
GROUP BY Department;
```

This groups employees by department and calculates statistics for each department independently.

Multiple Columns

```
SELECT Department, JobTitle,
       COUNT(*) as NumEmployees
FROM Employees
GROUP BY Department, JobTitle;
```

Key Rules

- Columns in SELECT must be in GROUP BY or be aggregate functions
- Can group by multiple columns
- NULL values form their own group
- GROUP BY processes data before ORDER BY

The HAVING Clause

HAVING filters groups after GROUP BY is applied, similar to how WHERE filters individual rows. Use HAVING for conditions involving aggregate functions.

WHERE vs HAVING

WHERE filters rows before grouping, while **HAVING** filters groups after aggregation. Use WHERE for column conditions and HAVING for aggregate conditions.

```
SELECT Department, AVG(Salary) as AvgSalary
FROM Employees
WHERE HireDate > '2020-01-01'  -- Filters rows first
GROUP BY Department
HAVING AVG(Salary) > 60000;     -- Filters groups
```

Practical Examples

HAVING is essential for finding groups that meet specific criteria based on their aggregate values.

```
-- Find departments with more than 5 employees
SELECT Department, COUNT(*) as EmpCount
FROM Employees
GROUP BY Department
HAVING COUNT(*) > 5;

-- Find products with average rating above 4.5
SELECT ProductID, AVG(Rating) as AvgRating
FROM Reviews
GROUP BY ProductID
HAVING AVG(Rating) > 4.5;
```

The ORDER BY Clause

ORDER BY sorts query results based on one or more columns, in ascending (ASC) or descending (DESC) order. This is always the last clause in a SELECT statement.

Basic Ordering

```
SELECT * FROM Students
ORDER BY LastName ASC;
```

Sorts results alphabetically by last name.

Multiple Columns

```
SELECT * FROM Students
ORDER BY Grade DESC,
      LastName ASC;
```

Primary sort by grade (highest first), secondary sort by name alphabetically.

Ordering with Aggregates

```
SELECT Department,
      COUNT(*) as NumEmp
FROM Employees
GROUP BY Department
ORDER BY NumEmp DESC;
```

Sorts departments by employee count in descending order.

Position Numbers

```
SELECT Name, Salary
FROM Employees
ORDER BY 2 DESC;
```

Can reference columns by position (2 = Salary column).

Combining GROUP BY, HAVING, and ORDER BY

These clauses work together to create sophisticated analytical queries that group, filter, and sort data in a single statement.

```
SELECT
    Department,
    JobTitle,
    COUNT(*) as EmployeeCount,
    AVG(Salary) as AvgSalary,
    MAX(Salary) as MaxSalary
FROM Employees
WHERE Status = 'Active'
GROUP BY Department, JobTitle
HAVING COUNT(*) >= 3
ORDER BY Department ASC, AvgSalary DESC;
```

01

WHERE filters rows

Selects only active employees before grouping

02

GROUP BY creates groups

Organizes data by department and job title combinations

03

Aggregate functions calculate

Computes count and salary statistics for each group

04

HAVING filters groups

Keeps only groups with at least 3 employees

05

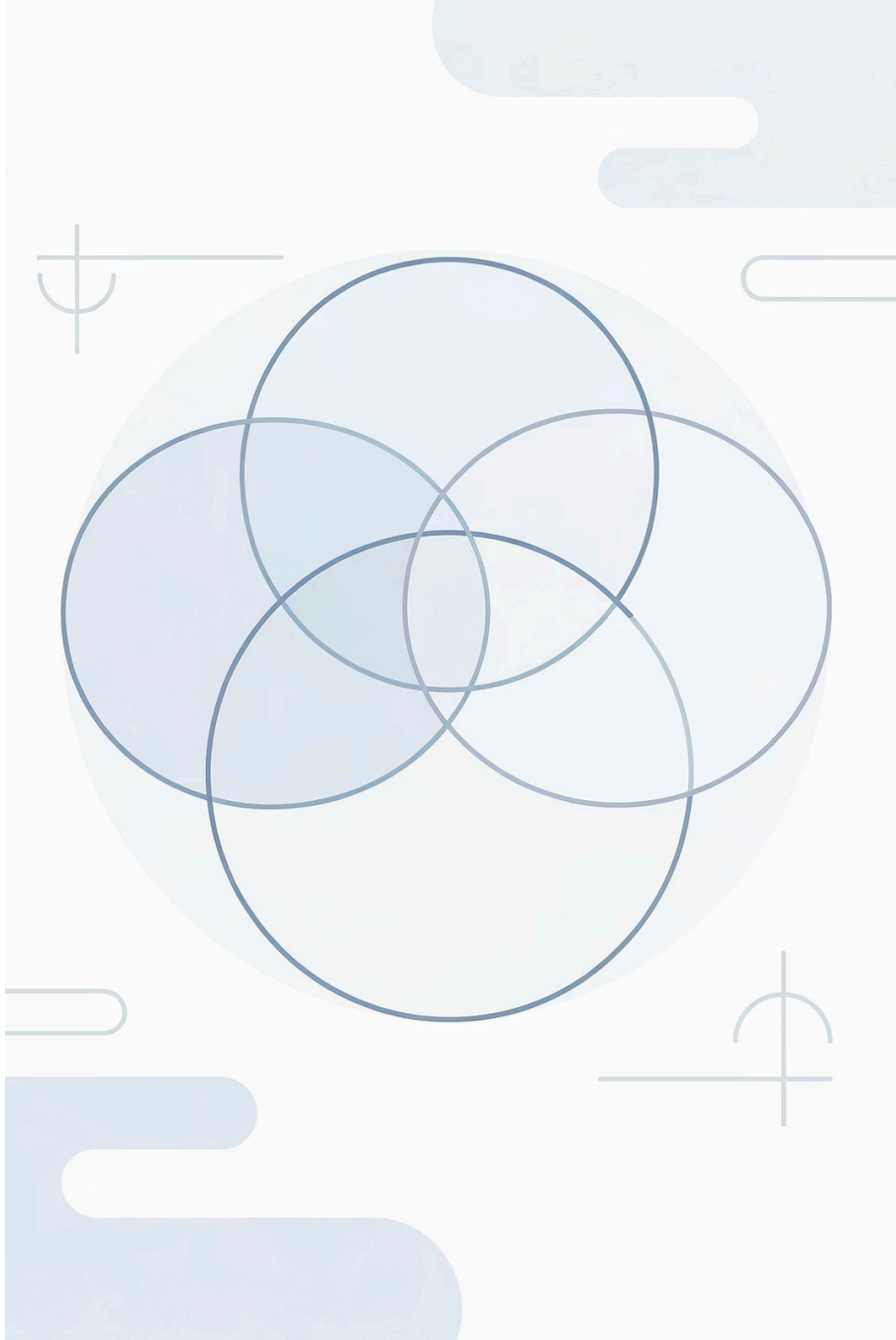
ORDER BY sorts results

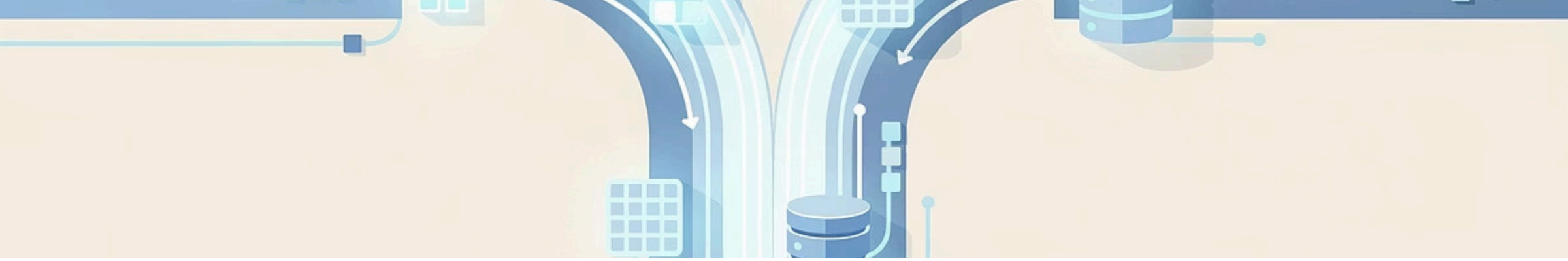
Arranges output by department and average salary

3.18

Set Operators

Set operators combine results from multiple SELECT statements, enabling complex queries that merge, compare, or exclude data from different sources. These operators treat query results as mathematical sets.





UNION and UNION ALL

UNION

Combines results from multiple queries and removes duplicate rows, returning only unique records.

```
SELECT StudentID, Name
FROM CurrentStudents
UNION
SELECT StudentID, Name
FROM FormerStudents;
```

- Removes duplicate rows automatically
- Requires same number of columns
- Column types must be compatible
- Slower than UNION ALL due to duplicate removal

UNION ALL

Combines all results from multiple queries, including duplicate rows, for faster performance.

```
SELECT ProductID, Name
FROM OnlineInventory
UNION ALL
SELECT ProductID, Name
FROM StoreInventory;
```

- Keeps all rows, including duplicates
- Faster than UNION
- Use when duplicates are acceptable or expected
- Same requirements as UNION

INTERSECT Operator

INTERSECT returns only rows that appear in both query results, identifying common records between two datasets.

Finding Common Elements

```
SELECT StudentID
FROM EnrolledInMath
INTERSECT
SELECT StudentID
FROM EnrolledInPhysics;
```

This query finds students enrolled in both Math and Physics courses, identifying the overlap between two student groups.

Practical Applications

- Finding customers who purchased from multiple categories
- Identifying employees with multiple certifications
- Locating products available in multiple warehouses
- Discovering shared interests or attributes

Important Notes

INTERSECT automatically removes duplicates from the result set. Both queries must return the same number of columns with compatible data types.

MINUS (or EXCEPT) Operator

MINUS returns rows from the first query that don't appear in the second query, effectively subtracting one result set from another.

Syntax and Usage

```
SELECT StudentID
FROM AllStudents
MINUS
SELECT StudentID
FROM GraduatedStudents;
```

This identifies current students by removing those who have graduated from the complete student list. Note: Some databases use EXCEPT instead of MINUS, but functionality is identical.

Real-World Scenarios

- **Inventory Management:** Find products ordered but not yet delivered
- **Customer Analysis:** Identify customers who haven't made recent purchases
- **Data Quality:** Locate records in one system missing from another
- **Access Control:** Find users with accounts but no assigned permissions

MINUS is particularly useful for data reconciliation and identifying gaps in datasets.

Set Operator Rules and Best Practices

1

Column Compatibility

Both queries must return the same number of columns, and corresponding columns must have compatible data types. Column names from the first query are used in results.

2

Performance Considerations

UNION ALL is faster than UNION because it doesn't remove duplicates. Use UNION ALL when duplicates are acceptable or when you know duplicates won't exist.

3

Ordering Results

ORDER BY can only be used at the end of the entire set operation, not within individual queries. Reference columns by position or name from first query.

4

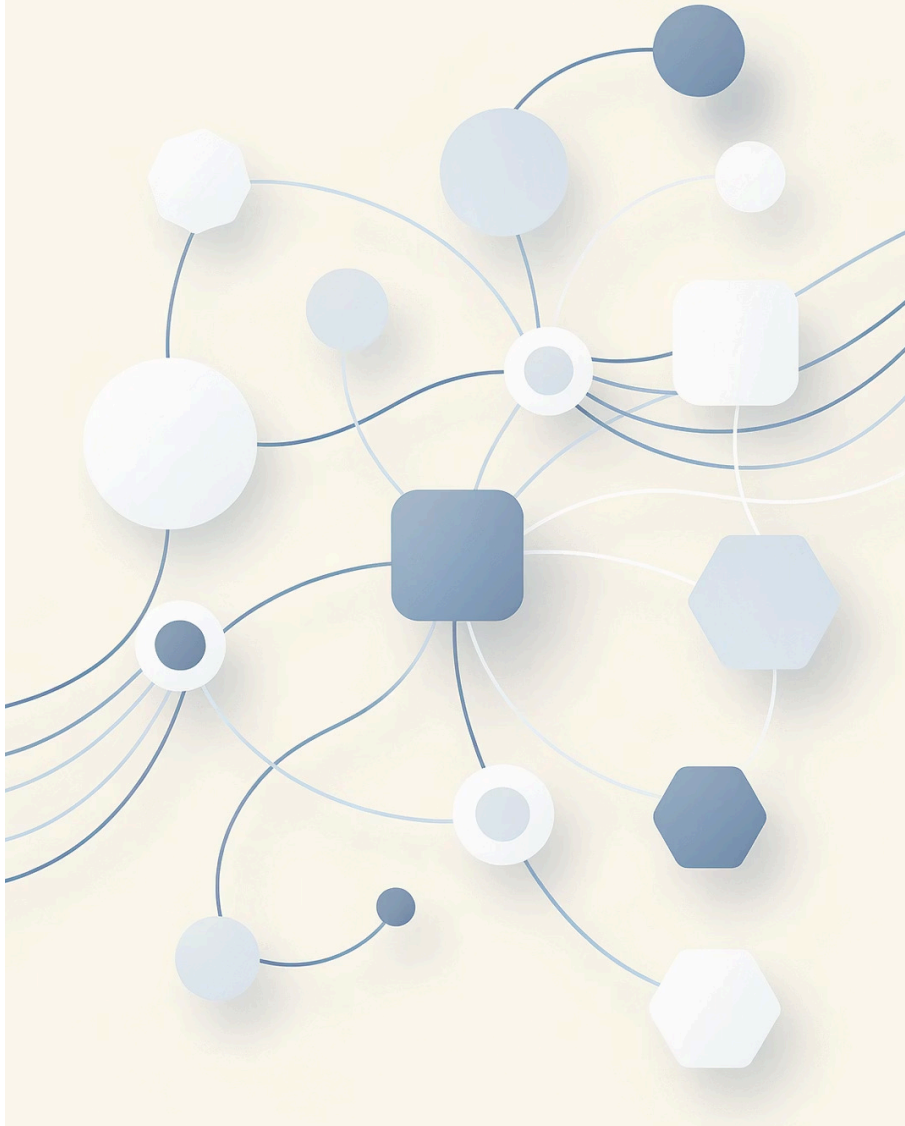
Parentheses for Clarity

When combining multiple set operators, use parentheses to control precedence and make your intentions clear to other developers.

3.19

SQL Joins

Joins are fundamental to relational databases, enabling you to retrieve related data from multiple tables in a single query. Understanding joins is essential for working with normalized database designs.



Simple Join (Cross Join)

A simple join, also called a Cartesian product or cross join, combines every row from the first table with every row from the second table.

Syntax

```
SELECT *
FROM Students, Courses;
```

or explicitly:

```
SELECT *
FROM Students
CROSS JOIN Courses;
```

When to Use

Cross joins are rarely used intentionally but can be useful for:

- Generating all possible combinations
- Creating test data
- Mathematical permutations

Warning: If Table A has 100 rows and Table B has 50 rows, a cross join produces 5,000 rows. Use cautiously to avoid performance issues.

Equi Join (Inner Join)

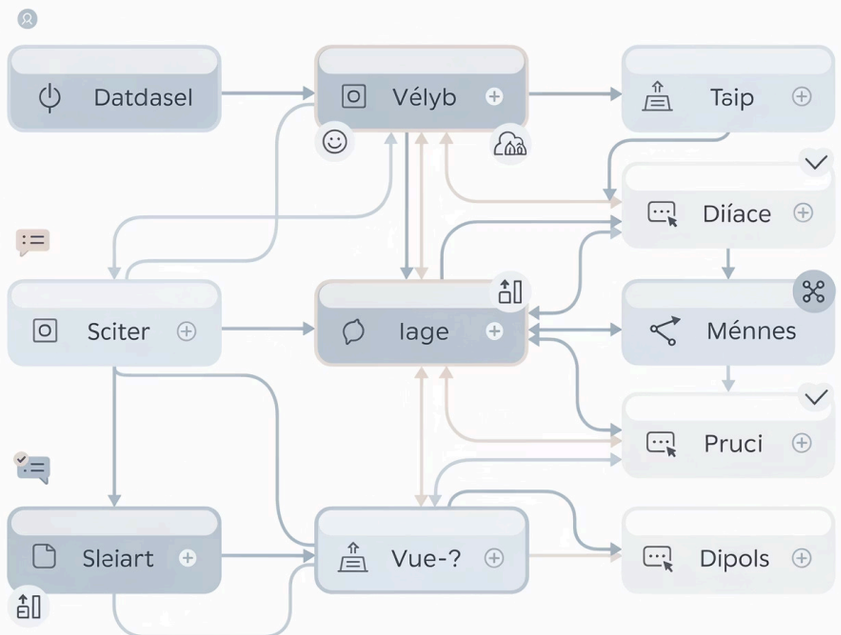
An equi join matches rows from two tables based on equality between specified columns. This is the most common type of join, returning only rows where the join condition is true.

Traditional Syntax

```
SELECT Students.Name,
      Enrollments.CourseID
FROM Students, Enrollments
WHERE Students.StudentID =
     Enrollments.StudentID;
```

ANSI Syntax (Preferred)

```
SELECT Students.Name,
      Enrollments.CourseID
FROM Students
INNER JOIN Enrollments
  ON Students.StudentID =
    Enrollments.StudentID;
```



The ANSI syntax is preferred because it separates join conditions from filter conditions, making queries more readable and maintainable.

Non-Equi Join

Non-equi joins use comparison operators other than equals (=) to match rows, useful for range-based relationships and fuzzy matching.

Range Matching

```
SELECT Employees.Name,
      SalaryGrades.Grade
FROM Employees
JOIN SalaryGrades
  ON Employees.Salary
    BETWEEN SalaryGrades.MinSalary
    AND SalaryGrades.MaxSalary;
```

Assigns salary grades to employees based on salary ranges defined in the SalaryGrades table.

Comparison Operators

Non-equi joins can use any comparison operator:

- **>** Greater than
- **<** Less than
- **>=** Greater than or equal
- **<=** Less than or equal
- **!=** Not equal
- **BETWEEN** Range checks

Use Cases

- Finding records within date ranges
- Matching prices to discount tiers
- Assigning grades based on score ranges
- Comparing timestamps for overlap detection

Self Join

A self join relates a table to itself, useful for hierarchical data or comparing rows within the same table.

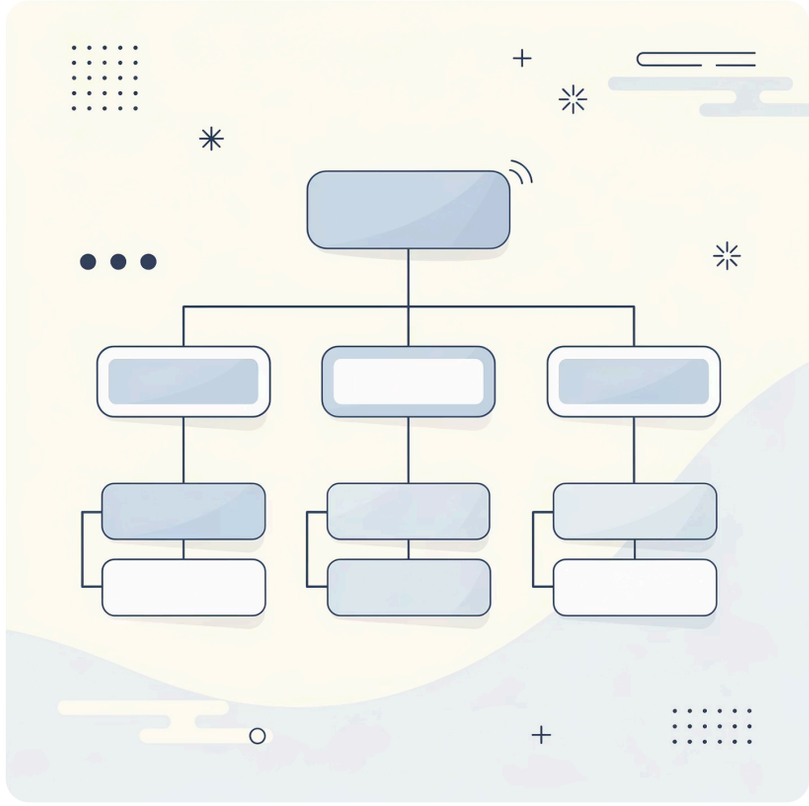
Finding Employee-Manager Relationships

```
SELECT
    E.EmployeeID,
    E.Name as EmployeeName,
    M.Name as ManagerName
FROM Employees E
JOIN Employees M
  ON E.ManagerID = M.EmployeeID;
```

This query uses table aliases (E and M) to reference the same Employees table twice, matching employees with their managers.

Other Self Join Applications

- Finding product pairs for recommendations
- Comparing sequential transactions
- Building organizational hierarchies
- Detecting duplicate or similar records



Always use meaningful table aliases with self joins to distinguish between the two instances of the same table.

Outer Joins

Outer joins return all rows from one or both tables, including rows without matching values in the other table. Unmatched columns contain NULL values.

LEFT OUTER JOIN

```
SELECT Students.Name,
      Enrollments.CourseID
FROM Students
LEFT JOIN Enrollments
  ON Students.StudentID =
    Enrollments.StudentID;
```

Returns all students, including those not enrolled in any courses. Non-enrolled students show NULL for CourseID.

RIGHT OUTER JOIN

```
SELECT Students.Name,
      Enrollments.CourseID
FROM Students
RIGHT JOIN Enrollments
  ON Students.StudentID =
    Enrollments.StudentID;
```

Returns all enrollments, including orphaned records where StudentID doesn't match any student (useful for data quality checks).

FULL OUTER JOIN

```
SELECT Students.Name,
      Enrollments.CourseID
FROM Students
FULL OUTER JOIN Enrollments
  ON Students.StudentID =
    Enrollments.StudentID;
```

Returns all students and all enrollments, with NULLs where matches don't exist on either side. Useful for comprehensive data analysis.

Join Best Practices

Use Appropriate Join Types

Select the join type that matches your data requirements. Inner joins for matching records only, outer joins when you need all records from one or both tables.

Optimize Join Conditions

Index columns used in join conditions to improve performance. Ensure joined columns have the same data type to avoid implicit conversions.

Use Table Aliases

Always use clear, meaningful aliases for tables, especially with self joins or complex queries involving many tables.

Test Your Logic

Verify join results with small datasets before running on production data. Pay special attention to NULL handling in outer joins.

3.20-3.23

Database Constraints

Constraints are rules enforced by the database system to maintain data accuracy, consistency, and integrity. They prevent invalid data from entering your database, acting as automated guardians of data quality.

The Need for Constraints

Why Constraints Matter

Without constraints, databases are vulnerable to data quality issues that can compromise business operations and decision-making. Constraints provide automatic validation that's more reliable than application-level checks.

- **Prevent Invalid Data:** Stop bad data at the database level
- **Enforce Business Rules:** Encode requirements directly in schema
- **Maintain Relationships:** Ensure referential integrity across tables
- **Improve Data Quality:** Reduce errors and inconsistencies

Types of Integrity

Domain Integrity

Ensures column values are valid and within acceptable ranges

Entity Integrity

Guarantees each row is unique and identifiable

Referential Integrity

Maintains valid relationships between tables

3.21

Domain Integrity Constraints

Domain constraints restrict the values that can be stored in a column, ensuring data conforms to business rules and logical requirements.

NOT NULL Constraint

The NOT NULL constraint ensures a column must contain a value—it cannot be left empty. This is essential for columns that are required for business operations.

Creating NOT NULL Columns

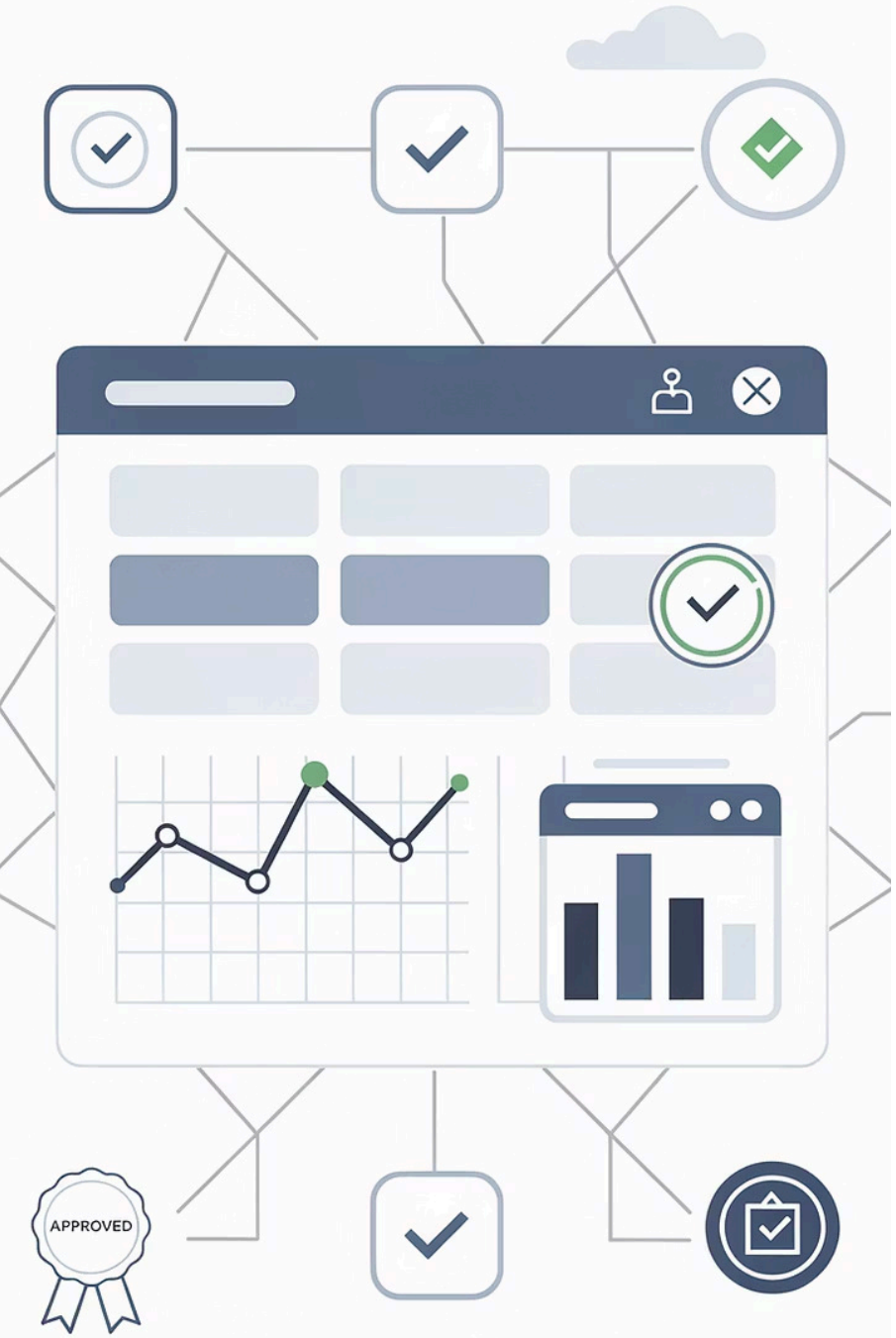
```
CREATE TABLE Students (
    StudentID INT NOT NULL,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    Email VARCHAR(100) NOT NULL,
    PhoneNumber VARCHAR(20)
);
```

In this example, student ID and name fields are mandatory, while phone number is optional.

When to Use NOT NULL

- Required identification fields
- Critical business data
- Foreign key columns
- Any field where missing data would cause problems

Attempting to insert or update a row with NULL in a NOT NULL column will result in an error, protecting data quality.



CHECK Constraint

CHECK constraints enforce specific conditions on column values, enabling complex business rules to be encoded directly in the database schema.

Range Validation

```
CREATE TABLE Products (
    ProductID INT,
    Price DECIMAL(10,2),
    Stock INT,
    CONSTRAINT chk_price
        CHECK (Price > 0),
    CONSTRAINT chk_stock
        CHECK (Stock >= 0)
);
```

Ensures prices are positive and stock levels are non-negative.

List Validation

```
CREATE TABLE Employees (
    EmployeeID INT,
    Status VARCHAR(20),
    CONSTRAINT chk_status
        CHECK (Status IN
        ('Active', 'Inactive',
         'On Leave'))
);
```

Restricts status to predefined valid values only.

Complex Conditions

```
CREATE TABLE Orders (
    OrderID INT,
    OrderDate DATE,
    ShipDate DATE,
    CONSTRAINT chk_dates
        CHECK (ShipDate >=
              OrderDate)
);
```

Ensures shipping date cannot be before order date, maintaining logical consistency.

3.22

Entity Integrity Constraints

Entity integrity ensures that each row in a table can be uniquely identified and that no two rows are identical. This is fundamental to relational database design.

UNIQUE Constraint

The UNIQUE constraint ensures all values in a column (or combination of columns) are distinct, preventing duplicate entries while allowing NULL values.

Single Column Unique

```
CREATE TABLE Students (
    StudentID INT,
    Email VARCHAR(100) UNIQUE,
    PhoneNumber VARCHAR(20) UNIQUE
);
```

Each student must have a unique email and phone number. Multiple NULL values are allowed because NULL is considered distinct from other NULLs.

Composite Unique

```
CREATE TABLE Enrollments (
    StudentID INT,
    CourseID INT,
    Semester VARCHAR(20),
    CONSTRAINT uc_enrollment
        UNIQUE (StudentID,
                CourseID,
                Semester)
);
```

UNIQUE vs PRIMARY KEY

- **UNIQUE:** Can have multiple per table, allows NULL values
- **PRIMARY KEY:** Only one per table, no NULL values
- Both prevent duplicate values
- Both automatically create indexes

Use UNIQUE for alternative identification fields like email addresses or social security numbers.

PRIMARY KEY Constraint

A PRIMARY KEY uniquely identifies each row in a table. It combines UNIQUE and NOT NULL constraints, serving as the definitive identifier for records.

Single Column Primary Key

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);
```

The most common scenario—a single column that uniquely identifies each record.

Composite Primary Key

```
CREATE TABLE OrderDetails (
    OrderID INT,
    ProductID INT,
    Quantity INT,
    PRIMARY KEY (OrderID,
                 ProductID)
);
```

Multiple columns combine to form a unique identifier, useful for junction tables.

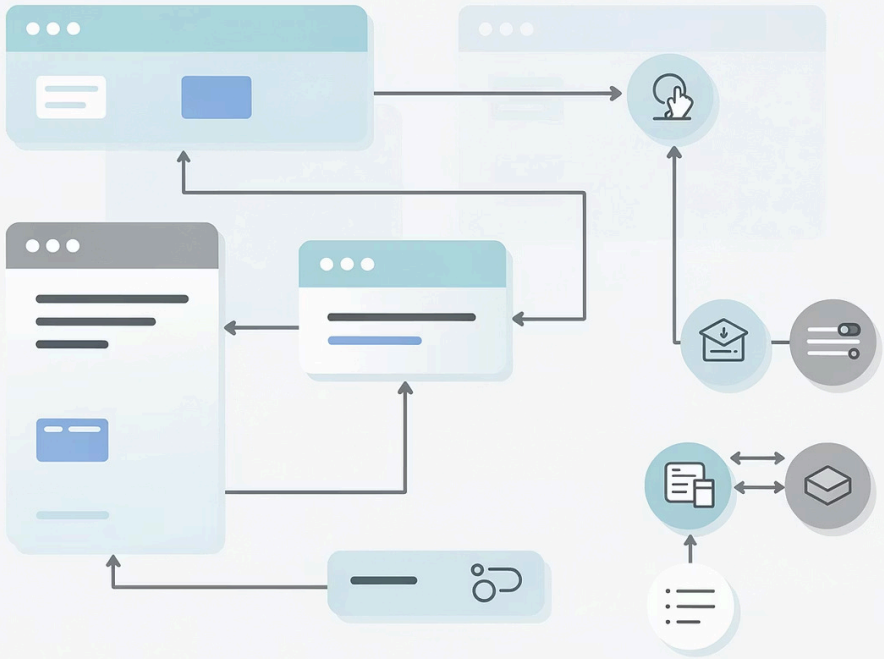
Named Primary Key

```
CREATE TABLE Courses (
    CourseID INT,
    CourseName VARCHAR(100),
    CONSTRAINT pk_courses
        PRIMARY KEY (CourseID)
);
```

Naming constraints makes them easier to reference and maintain.

Primary Key Best Practices

- Every table should have a primary key
- Choose simple, stable values that won't change
- Consider using auto-increment integers for simplicity
- Avoid using business data (like email) as primary keys



3.23

Referential Integrity Constraints

Referential integrity maintains valid relationships between tables, ensuring that references to data in other tables are always valid and consistent.

FOREIGN KEY and REFERENCE KEY Constraints

Foreign keys create links between tables, enforcing relationships and preventing orphaned records. They ensure that values in one table correspond to valid entries in another table.

Basic Foreign Key

```
CREATE TABLE Enrollments (
    EnrollmentID INT PRIMARY KEY,
    StudentID INT,
    CourseID INT,
    CONSTRAINT fk_student
        FOREIGN KEY (StudentID)
        REFERENCES Students(StudentID),
    CONSTRAINT fk_course
        FOREIGN KEY (CourseID)
        REFERENCES Courses(CourseID)
);
```

This ensures every enrollment references valid students and courses.

Referential Actions

- **ON DELETE CASCADE:** Delete child records when parent is deleted
- **ON DELETE SET NULL:** Set foreign key to NULL when parent is deleted
- **ON UPDATE CASCADE:** Update foreign key when parent key changes
- **ON DELETE RESTRICT:** Prevent parent deletion if children exist

```
FOREIGN KEY (StudentID)
REFERENCES Students(StudentID)
ON DELETE CASCADE
ON UPDATE CASCADE;
```

1

Parent Table

Contains the primary key being referenced

2

Child Table

Contains the foreign key column

∞

Relationship

One parent can have many children

Referential integrity is the foundation of relational database design, enabling complex data relationships while maintaining consistency. It prevents data anomalies and ensures your database accurately represents real-world relationships. Understanding and properly implementing these constraints is essential for creating robust, reliable database applications.