SQL: Definitive Guide from Beginner to Expert-Level Topics

SQL (Structured Query Language) is the standard language for managing and querying data in relational database management systems (RDBMS) like MySQL, PostgreSQL, SQLite, Oracle, and SQL Server. Used since the 1970s, SQL powers applications from small apps to enterprise systems, enabling data storage, retrieval, and analysis. This README is the ultimate guide to SQL, covering every topic from beginner to expert level with exhaustive explanations, practical examples, edge cases, and cross-connections, matching the depth of a Rust README.

Table of Contents

Introduction to SQL

SQL, developed in the 1970s by IBM, is a declarative language for interacting with relational databases. Its key features include:

- **Relational Model**: Organizes data into tables with rows and columns, linked by keys.
- **Declarative Syntax**: Specify *what* to retrieve, not *how*.
- **Standardization**: ANSI/ISO standards ensure portability across RDBMS.

- **Multi-Purpose**: Supports data definition (DDL), manipulation (DML), querying (DQL), and control (DCL).
- **Use Cases**: Web apps, data analytics, reporting, ETL pipelines, and more.

SQL's strength lies in its simplicity for basic queries and power for complex data analysis, making it essential for developers, data analysts, and DBAs.

Getting Started: First Query

To begin, install an RDBMS (e.g., MySQL, PostgreSQL, SQLite). SQLite is lightweight and file-based, ideal for learning:

```
# Install SQLite (Ubuntu)
sudo apt install sqlite3
# Or download from https://www.sqlite.org/download.html
```

Create a database and run a query:

```
-- Open SQLite CLI
sqlite3 mydb.db

-- Create a table
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL
);

-- Insert data
INSERT INTO users (name) VALUES ('Alice');

-- Query data
SELECT * FROM users;
```

Output:

```
id | name
1  | Alice
```

Connect via GUI tools (e.g., DBeaver, pgAdmin) or programmatically (e.g., Python's `sqlite3`, PHP's PDO).

SQL Standards and Variants

SQL is governed by ANSI/ISO standards (e.g., SQL-92, SQL:2016), but RDBMS vendors implement extensions:

- **MySQL/MariaDB**: JSON functions, `INSERT ... ON DUPLICATE KEY UPDATE`.
- **PostgreSQL**: Advanced CTEs, window functions, full-text search.
- **SQLite**: Lightweight, supports most SQL-92, lacks stored procedures.
- **SQL Server**: T-SQL, `MERGE`, temporal tables.
- **Oracle**: PL/SQL, hierarchical queries.

Key differences:

- **Data Types**: `VARCHAR` vs. `TEXT`, `BIGINT` vs. `NUMBER`.
- **Functions**: `CONCAT` vs. `||`, `NOW()` vs. `CURRENT_TIMESTAMP`.
- **Limits**: SQLite's no `RIGHT JOIN`, MySQL's no `EXCEPT`.

This guide uses standard SQL, noting vendor-specific syntax where relevant.

Check version:

```
-- MySQL
SELECT VERSION();
-- PostgreSQL
SELECT version();
-- SQLite
SELECT sqlite_version();
```

Basic Concepts

Database and Schema

A **database** is a container for data; a **schema** organizes objects (tables, views) within it.

```
-- Create database (MySQL/PostgreSQL)
CREATE DATABASE myapp;

-- Create schema (PostgreSQL/SQL Server)
CREATE SCHEMA app;

-- Use database
USE myapp; -- MySQL
\c myapp; -- PostgreSQL psql
```

Cross-Connections:

- **Tables**: Reside in schemas.
- **Security**: Schemas control access.
- **Performance**: Partition data across schemas.

Tables and Data Types

Tables store data in rows and columns with defined types:

- **Numeric**: `INTEGER`, `BIGINT`, `DECIMAL`, `FLOAT`.
- **String**: `CHAR(n)`, `VARCHAR(n)`, `TEXT`.
- **Date/Time**: `DATE`, `TIMESTAMP`, `INTERVAL` (PostgreSQL).
- **Boolean**: `BOOLEAN` (PostgreSQL), `TINYINT(1)` (MySQL).
- **Binary**: `BLOB`, `BYTEA` (PostgreSQL).
- **Special**: `JSON`, `UUID` (PostgreSQL), `GEOMETRY` (MySQL/PostgreSQL).

```
CREATE TABLE employees (
    id INTEGER PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    salary DECIMAL(10, 2),
    hire_date DATE,
    active BOOLEAN DEFAULT TRUE
);
```

Edge Cases:

- **Type Limits**: `VARCHAR(255)` vs. `TEXT` storage overhead.
- **Implicit Casting**: `WHERE id = '1'` may cast string to integer.
- **Vendor Differences**: `SERIAL` (PostgreSQL) vs. `AUTO_INCREMENT` (MySQL).

Cross-Connections:

- **Constraints**: Enforce data integrity.
- **Indexes**: Optimize type-based queries.
- **JSON**: Store semi-structured data.

Basic Queries

Retrieve data with `SELECT`:

```
SELECT id, name FROM employees WHERE salary > 50000 ORDER BY hire_date DESC LIMIT 10;
```

Components:

- **SELECT**: Columns or expressions.
- **FROM**: Source table(s).
- **WHERE**: Filter rows.
- **ORDER BY**: Sort results.
- **LIMIT/OFFSET**: Paginate (vendor-specific: `TOP` in SQL Server, `FETCH FIRST` in Oracle).

Cross-Connections:

- **JOIN**: Combine tables.
- **Aggregates**: Summarize data.
- **Subqueries**: Nest queries.

Data Manipulation

INSERT

Add rows to tables:

```
-- Single row
INSERT INTO employees (id, name, salary, hire_date)
VALUES (1, 'Alice', 60000.00, '2023-01-15');
```

```
-- Multiple rows
INSERT INTO employees (id, name, salary)
VALUES (2, 'Bob', 55000.00), (3, 'Charlie', 65000.00);

-- Insert from query
INSERT INTO employees (id, name, salary)
SELECT id + 100, name, salary * 1.1 FROM employees WHERE active = TRUE;
```

Advanced Features:

- **RETURNING**: (PostgreSQL) Retrieve inserted rows: `INSERT ... RETURNING id`.
- **ON DUPLICATE KEY UPDATE**: (MySQL) Update if key exists.
- **DEFAULT Values**: Omit columns for defaults.

Edge Cases:

- **Constraint Violations**: Duplicate keys or NULL in `NOT NULL`.
- **Auto-Increment**: Skipping IDs can fragment sequences.
- **Bulk Inserts**: Large inserts may require batching for performance.

Cross-Connections:

- **Constraints**: Enforce data rules.
- **Transactions**: Ensure atomicity.
- **Triggers**: React to inserts.

UPDATE

Modify existing rows:

```
UPDATE employees
SET salary = salary * 1.05, active = TRUE
WHERE hire_date < '2024-01-01';
```

Advanced Features:

- **FROM Clause**: (PostgreSQL/SQL Server) Update with joins.
- **RETURNING**: (PostgreSQL) Retrieve updated rows.
- **CASE**: Conditional updates:

```
UPDATE employees
SET salary = CASE
    WHEN salary < 50000 THEN salary * 1.1
    ELSE salary * 1.05
END;
```

Edge Cases:

- **No Rows Updated**: `WHERE` mismatch returns 0 rows affected.

- **Deadlocks**: Concurrent updates may lock rows.
- **NULL Handling**: `SET column = NULL` vs. omitting column.

Cross-Connections:

- **JOIN**: Update with related data.
- **Triggers**: Log updates.
- **Indexes**: Optimize `WHERE` clauses.

DELETE

Remove rows:

```
DELETE FROM employees WHERE active = FALSE;
```

Advanced Features:

- **USING**: (PostgreSQL) Delete with joins.
- **RETURNING**: (PostgreSQL) Retrieve deleted rows.
- **TRUNCATE**: Fast, non-logged deletion (no triggers).

```
TRUNCATE TABLE employees RESTART IDENTITY; -- Resets auto-increment
```

Edge Cases:

- **Foreign Key Violations**: Cascading deletes required.
- **Performance**: Large deletes may lock tables; use batches.
- **No Undo**: `DELETE` is permanent unless in a transaction.

Cross-Connections:

- **Constraints**: Foreign keys affect deletes.
- **Transactions**: Roll back deletes.
- **Triggers**: Fire on deletes.

Querying Data

SELECT

Retrieve specific columns or computed values:

```
SELECT name, salary * 1.1 AS projected_salary
FROM employees
WHERE hire_date >= '2023-01-01';
```

Advanced Features:

- **Expressions**: `SELECT UPPER(name), salary + 1000`.
- **DISTINCT**: Remove duplicates: `SELECT DISTINCT department`.

- **COALESCE**: Handle NULLs: `SELECT COALESCE(salary, 0)`.

Edge Cases:

- **NULL in Expressions**: `NULL + 1 = NULL`.
- **Alias Conflicts**: Avoid reserved words in aliases.
- **Performance**: Select only needed columns to reduce I/O.

Cross-Connections:

- **Aggregates**: Compute summaries.
- **JOIN**: Combine data.
- **Subqueries**: Nest results.

WHERE

Filter rows with conditions:

```
SELECT * FROM employees
WHERE salary BETWEEN 50000 AND 70000
AND name LIKE 'A%'
AND hire_date IS NOT NULL;
```

Advanced Features:

- **IN**: `WHERE department IN ('HR', 'IT')`.
- **EXISTS**: Correlated subqueries: `WHERE EXISTS (SELECT 1 FROM ... )`.
- **REGEXP**: (MySQL/PostgreSQL) Pattern matching.

Edge Cases:

- **NULL Comparisons**: `column = NULL` is false; use `IS NULL`.
- **Short-Circuiting**: `WHERE FALSE AND slow_function()` may still evaluate.
- **Index Usage**: Complex `WHERE` may skip indexes.

Cross-Connections:

- **Indexes**: Optimize filters.
- **JOIN**: Filter joined rows.
- **Performance**: Analyze `WHERE` efficiency.

JOIN

Combine tables based on keys:

```
SELECT e.name, d.department_name
FROM employees e
INNER JOIN departments d ON e.department_id = d.id
LEFT JOIN bonuses b ON e.id = b.employee_id
WHERE b.amount IS NULL;
```

Join Types:

- **INNER JOIN**: Matching rows only.
- **LEFT/RIGHT JOIN**: Include all rows from one table.
- **FULL JOIN**: All rows, with NULLs for non-matches.
- **CROSS JOIN**: Cartesian product.

Advanced Features:

- **NATURAL JOIN**: Match on same-named columns (avoid due to ambiguity).
- **LATERAL**: (PostgreSQL) Subquery per row.
- **Multiple Joins**: Chain joins for complex queries.

Edge Cases:

- **Ambiguous Columns**: Qualify with table aliases (e.g., `e.id`).
- **NULL in Joins**: `ON column = NULL` excludes rows; use `IS NULL`.
- **Performance**: Joins on unindexed columns are slow.

Cross-Connections:

- **Indexes**: Optimize join keys.
- **Subqueries**: Alternative to joins.
- **Views**: Simplify complex joins.

GROUP BY and HAVING

Aggregate data by groups:

```
SELECT department_id, COUNT(*) AS emp_count, AVG(salary) AS avg_salary
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 5
ORDER BY avg_salary DESC;
```

Aggregates:

- **COUNT**, **SUM**, **AVG**, **MIN**, **MAX**.
- **Vendor-Specific**: `GROUP_CONCAT` (MySQL), `STRING_AGG` (PostgreSQL).

Advanced Features:

- **ROLLUP/CUBE**: (PostgreSQL/SQL Server) Subtotals.
- **GROUPING SETS**: Multiple groupings in one query.
- **Non-Aggregated Columns**: Must appear in `GROUP BY` or be aggregated.

Edge Cases:

- **NULL Groups**: `GROUP BY column` treats NULLs as one group.
- **HAVING vs. WHERE**: `HAVING` filters groups, not rows.
- **Vendor Quirks**: MySQL allows non-standard `GROUP BY` behavior (pre-8.0).

Cross-Connections:

- **Window Functions**: Alternative for aggregates.
- **JOIN**: Group across multiple tables.
- **Performance**: Indexes on `GROUP BY` columns.

ORDER BY

Sort results:

```
SELECT name, salary
FROM employees
ORDER BY salary DESC, name ASC NULLS LAST;
```

Advanced Features:

- **Expressions**: `ORDER BY LENGTH(name)`.
- **NULLS FIRST/LAST**: (PostgreSQL, some others).
- **Dynamic Sorting**: Use `CASE` in `ORDER BY`.

Edge Cases:

- **Performance**: Sorting large datasets requires indexes.
- **Collation**: Case-sensitive vs. case-insensitive sorting.
- **Non-Deterministic**: Without `ORDER BY`, row order is undefined.

Cross-Connections:

- **Indexes**: Optimize sorting.
- **LIMIT**: Combine for pagination.
- **Window Functions**: Use with ranking.

Data Definition

CREATE TABLE

Define tables with columns and constraints:

```
CREATE TABLE tasks (
    id INTEGER PRIMARY KEY AUTO_INCREMENT, -- MySQL
    description TEXT NOT NULL,
    status VARCHAR(20) DEFAULT 'pending',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

PostgreSQL Variant:

```
CREATE TABLE tasks (
    id SERIAL PRIMARY KEY,
    description TEXT NOT NULL,
    status VARCHAR(20) DEFAULT 'pending',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Cross-Connections:

- **Constraints**: Enforce rules.
- **Indexes**: Add for performance.
- **Triggers**: React to changes.

ALTER TABLE

Modify table structure:

```
ALTER TABLE tasks
ADD priority INTEGER DEFAULT 0,
DROP COLUMN status,
ALTER COLUMN description SET NOT NULL;
```

Advanced Features:

- **ADD CONSTRAINT**: Add keys or checks.
- **RENAME**: Change table/column names.
- **Partitioning**: (PostgreSQL/MySQL) Split large tables.

Edge Cases:

- **Data Loss**: Dropping columns or constraints.
- **Locking**: `ALTER` may lock tables in production.
- **Vendor Limits**: SQLite has limited `ALTER` support.

Cross-Connections:

- **Constraints**: Modify rules.
- **Indexes**: Rebuild after changes.
- **Performance**: Analyze impact.

DROP TABLE

Remove tables:

```
DROP TABLE tasks CASCADE; -- Drops dependent objects
```

Edge Cases:

- **Dependencies**: Foreign keys block drops unless `CASCADE`.

- **Irreversible**: No undo without backups.
- **Performance**: Dropping large tables may be slow.

Cross-Connections:

- **Constraints**: Remove dependent keys.
- **Transactions**: Drop in transactions for safety.
- **Views**: Invalidate dependent views.

Constraints

Enforce data integrity:

- **PRIMARY KEY**: Unique identifier.
- **FOREIGN KEY**: Link tables.
- **UNIQUE**: Ensure unique values.
- **NOT NULL**: Require values.
- **CHECK**: Custom rules (e.g., `salary > 0`).

```
CREATE TABLE orders (
    id INTEGER PRIMARY KEY,
    customer_id INTEGER,
    amount DECIMAL(10, 2) CHECK (amount >= 0),
    FOREIGN KEY (customer_id) REFERENCES customers(id) ON DELETE CASCADE
);
```

Advanced Features:

- **DEFERRABLE**: (PostgreSQL) Delay constraint checks until transaction end.
- **ON UPDATE/DELETE**: `CASCADE`, `SET NULL`, `RESTRICT`.
- **Composite Keys**: Multiple columns in `PRIMARY KEY`.

Edge Cases:

- **Circular References**: Require `DEFERRABLE` or careful design.
- **Performance**: Constraints add overhead.
- **NULL in UNIQUE**: Allowed in most RDBMS (except SQL Server).

Cross-Connections:

- **Indexes**: Automatically created for keys.
- **Triggers**: Alternative for complex rules.
- **Transactions**: Ensure constraint consistency.

Advanced Querying

Subqueries

Nest queries within queries:

```
SELECT name
FROM employees
WHERE department_id IN (
    SELECT id FROM departments WHERE location = 'HQ'
);
```

Types:

- **Scalar**: Single value: `SELECT (SELECT MAX(salary) FROM employees)`.
- **Row**: Multiple columns: `WHERE (id, name) = (SELECT 1, 'Alice')`.
- **Correlated**: References outer query.

```
SELECT e.name
FROM employees e
WHERE EXISTS (
    SELECT 1 FROM bonuses b WHERE b.employee_id = e.id AND b.amount > 1000
);
```

Edge Cases:

- **Performance**: Correlated subqueries can be slow; use joins if possible.
- **NULL Handling**: Subqueries returning NULL affect results.
- **Multiple Rows**: Scalar subqueries must return one row or error.

Cross-Connections:

- **JOIN**: Often replace subqueries.
- **CTEs**: Improve readability.
- **Performance**: Optimize with indexes.

Common Table Expressions (CTEs)

Define temporary result sets:

```
WITH high_earners AS (
    SELECT id, name, salary
    FROM employees
    WHERE salary > 70000
)
SELECT h.name, d.department_name
FROM high_earners h
JOIN departments d ON h.department_id = d.id;
```

Advanced Features:

- **Recursive CTEs**: (PostgreSQL, SQL Server) For hierarchical data.

```
WITH RECURSIVE org_chart AS (
    SELECT id, name, manager_id, 1 AS level
    FROM employees
    WHERE manager_id IS NULL
    UNION ALL
    SELECT e.id, e.name, e.manager_id, o.level + 1
    FROM employees e
    JOIN org_chart o ON e.manager_id = o.id
)
SELECT name, level
FROM org_chart
ORDER BY level;
```

- **Writable CTEs**: (PostgreSQL) Update/delete within CTEs.

Edge Cases:

- **Recursion Limits**: Prevent infinite loops.
- **Performance**: Recursive CTEs may be slower than iterative solutions.
- **Vendor Support**: MySQL 8.0+ and SQLite 3.8.3+ support CTEs.

Cross-Connections:

- **Subqueries**: Alternative syntax.
- **Window Functions**: Combine for analytics.
- **Views**: Persistent CTE equivalents.

Window Functions

Perform calculations across rows:

```
SELECT
    name,
    salary,
    RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS rank,
    SUM(salary) OVER (PARTITION BY department_id) AS dept_total
FROM employees;
```

Functions:

- **Ranking**: `RANK`, `DENSE_RANK`, `ROW_NUMBER`.
- **Aggregates**: `SUM`, `AVG`, `COUNT` over windows.
- **Value**: `LAG`, `LEAD`, `FIRST_VALUE`.

Advanced Features:

- **Frame Specification**: `ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING`.
- **OVER Clause**: Define partitions and ordering.
- **Vendor-Specific**: PostgreSQL's `FILTER` in aggregates.

Edge Cases:

- **NULL Handling**: `LAG` returns NULL at boundaries.
- **Performance**: Window functions scan large datasets.
- **Order Sensitivity**: Ties in `RANK` vs. `DENSE_RANK`.

Cross-Connections:

- **GROUP BY**: Complementary for aggregates.
- **Indexes**: Optimize `ORDER BY` in windows.
- **CTEs**: Combine for complex analytics.

UNION, INTERSECT, EXCEPT

Combine query results:

```
SELECT name FROM employees WHERE salary > 60000
UNION
SELECT name FROM contractors WHERE rate > 100
ORDER BY name;

SELECT department_id FROM employees
INTERSECT
SELECT department_id FROM projects;
```

Advanced Features:

- **UNION ALL**: Faster, includes duplicates.
- **EXCEPT**: (PostgreSQL/SQL Server) Rows in first query but not second.
- **Column Matching**: Same number and compatible types.

Edge Cases:

- **Performance**: `UNION` removes duplicates, slower than `UNION ALL`.
- **Type Coercion**: Implicit casting may occur.
- **ORDER BY**: Applies to final result, not individual queries.

Cross-Connections:

- **Subqueries**: Combine with sets.
- **JOIN**: Alternative for some `INTERSECT` cases.
- **Views**: Store combined results.

Indexes and Performance

Index Types

Indexes speed up queries:

- **B-Tree**: Default for most columns.

- **Hash**: Equality checks (MySQL/PostgreSQL).
- **GiST/GIN**: (PostgreSQL) Full-text search, JSON.
- **Clustered**: (SQL Server/MySQL) Physical row order.
- **Composite**: Multiple columns: `CREATE INDEX idx ON employees (department_id, salary)`.

```
CREATE INDEX idx_salary ON employees (salary);
CREATE UNIQUE INDEX idx_email ON employees (email);
```

Cross-Connections:

- **Constraints**: Keys create indexes.
- **Performance**: Indexes optimize `WHERE`/`JOIN`.
- **Triggers**: Indexes on updated columns.

Query Optimization

Improve query performance:

- **Indexes**: Cover `WHERE`, `JOIN`, `ORDER BY` columns.
- **Avoid Functions on Columns**: `WHERE UPPER(name) = 'ALICE'` skips indexes.
- **Denormalization**: Reduce joins for read-heavy apps.
- **Partitioning**: Split large tables (PostgreSQL/MySQL).

Edge Cases:

- **Over-Indexing**: Slows writes, increases storage.
- **Index Bloat**: Rebuild indexes periodically.
- **Statistics**: Update stats for query planner (e.g., `ANALYZE`).

Cross-Connections:

- **EXPLAIN**: Analyze query plans.
- **Transactions**: Locks affect performance.
- **Views**: Optimize underlying queries.

EXPLAIN Plans

Analyze query execution:

```
-- PostgreSQL
EXPLAIN ANALYZE SELECT * FROM employees WHERE salary > 50000;

-- MySQL
EXPLAIN SELECT * FROM employees WHERE salary > 50000;
```

Key Terms:

- **Seq Scan**: Full table scan.

- **Index Scan**: Uses index.
- **Cost**: Estimated resource usage.
- **Rows**: Estimated rows returned.

Edge Cases:

- **Planner Errors**: Outdated stats lead to bad plans.
- **Vendor Differences**: Syntax and output vary.
- **Performance**: `EXPLAIN ANALYZE` runs query, impacting production.

Cross-Connections:

- **Indexes**: Verify usage.
- **Query Tuning**: Rewrite based on plans.
- **Performance**: Optimize bottlenecks.

Transactions

ACID Properties

Transactions ensure:

- **Atomicity**: All or nothing.
- **Consistency**: Constraints preserved.
- **Isolation**: Partial changes invisible.
- **Durability**: Committed changes persist.

```
BEGIN;
INSERT INTO accounts (id, balance) VALUES (1, 1000);
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
```

Cross-Connections:

- **Constraints**: Enforced during transactions.
- **Triggers**: Fire within transactions.
- **Performance**: Long transactions lock resources.

COMMIT and ROLLBACK

Control transaction outcomes:

```
BEGIN;
INSERT INTO accounts (id, balance) VALUES (3, -100);
-- Error: CHECK constraint violation
ROLLBACK;
```

Advanced Features:

- **SAVEPOINT**: Partial rollbacks (PostgreSQL/MySQL).
- **AUTOCOMMIT**: Enabled by default in some RDBMS.
- **Two-Phase Commit**: (PostgreSQL) Distributed transactions.

Edge Cases:

- **Deadlocks**: Concurrent transactions may abort.
- **Long Transactions**: Increase lock contention.
- **Connection Loss**: Implicit rollback in most RDBMS.

Cross-Connections:

- **Error Handling**: Catch errors to rollback.
- **Indexes**: Minimize locks.
- **Security**: Transactions prevent race conditions.

Isolation Levels

Control visibility of changes:

- **Read Uncommitted**: See uncommitted changes (rarely used).
- **Read Committed**: See committed changes only.
- **Repeatable Read**: Consistent reads, but phantom rows possible.
- **Serializable**: Full isolation, slowest.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;
SELECT * FROM accounts WHERE balance > 0;
-- Other transaction can't insert conflicting rows
COMMIT;
```

Edge Cases:

- **Performance**: Higher isolation slows concurrency.
- **Phantom Reads**: Possible in `Repeatable Read`.
- **Vendor Defaults**: MySQL uses `Repeatable Read`, PostgreSQL uses `Read Committed`.

Cross-Connections:

- **Transactions**: Isolation defines behavior.
- **Locks**: Higher isolation increases locking.
- **Performance**: Balance isolation and speed.

Stored Procedures and Functions

Stored Procedures

Encapsulate logic in the database:

```
-- PostgreSQL
CREATE PROCEDURE update_salary(rate DECIMAL)
LANGUAGE SQL
AS $$
    UPDATE employees
    SET salary = salary * rate
    WHERE active = TRUE;
$$;

CALL update_salary(1.05);

-- MySQL
DELIMITER //
CREATE PROCEDURE update_salary(IN rate DECIMAL(5,2))
BEGIN
    UPDATE employees
    SET salary = salary * rate
    WHERE active = TRUE;
END //
DELIMITER ;

CALL update_salary(1.05);
```

Cross-Connections:

- **Triggers**: Similar but event-driven.
- **Security**: Restrict direct table access.
- **Performance**: Reduce network round-trips.

User-Defined Functions

Return values for queries:

```
-- PostgreSQL
CREATE FUNCTION calculate_bonus(salary DECIMAL)
RETURNS DECIMAL
AS $$
    SELECT salary * 0.1;
$$ LANGUAGE SQL;

SELECT name, calculate_bonus(salary) AS bonus FROM employees;

-- MySQL
DELIMITER //
CREATE FUNCTION calculate_bonus(salary DECIMAL(10,2))
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
    RETURN salary * 0.1;
```

```
END //
DELIMITER ;

SELECT name, calculate_bonus(salary) AS bonus FROM employees;
```

Advanced Features:

- **Table Functions**: (PostgreSQL) Return row sets.
- **Deterministic**: Optimize caching (MySQL).
- **PL/pgSQL, T-SQL**: Procedural languages for complex logic.

Edge Cases:

- **Performance**: Functions may be slower than native SQL.
- **Security**: Functions run with definer/invoker privileges.
- **Vendor Support**: SQLite lacks stored procedures.

Cross-Connections:

- **Views**: Combine with functions.
- **Triggers**: Call functions.
- **Performance**: Inline simple functions.

Triggers

Execute code on events:

```
-- PostgreSQL
CREATE FUNCTION log_salary_change()
RETURNS TRIGGER
AS $$
BEGIN
    INSERT INTO salary_audit (employee_id, old_salary, new_salary, changed_at)
    VALUES (OLD.id, OLD.salary, NEW.salary, CURRENT_TIMESTAMP);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER salary_trigger
AFTER UPDATE OF salary ON employees
FOR EACH ROW
EXECUTE FUNCTION log_salary_change();

-- MySQL
DELIMITER //
CREATE TRIGGER salary_trigger
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO salary_audit (employee_id, old_salary, new_salary, changed_at)
    VALUES (OLD.id, OLD.salary, NEW.salary, NOW());
```

```
END //
DELIMITER ;
```

Advanced Features:

- **INSTEAD OF**: (PostgreSQL) For views.
- **Statement Triggers**: Fire once per statement.
- **Conditional Triggers**: `WHEN` clause for filtering.

Edge Cases:

- **Recursion**: Triggers calling triggers can loop.
- **Performance**: Triggers add overhead to DML.
- **Debugging**: Hard to trace in complex setups.

Cross-Connections:

- **Constraints**: Alternative for simple rules.
- **Stored Procedures**: Share logic.
- **Security**: Audit changes.

Views and Materialized Views

Views are virtual tables; materialized views store data.

```
CREATE VIEW active_employees AS
SELECT id, name, salary
FROM employees
WHERE active = TRUE;

SELECT * FROM active_employees WHERE salary > 50000;
```

Materialized View (PostgreSQL):

```
CREATE MATERIALIZED VIEW dept_stats AS
SELECT department_id, COUNT(*) AS emp_count, AVG(salary) AS avg_salary
FROM employees
GROUP BY department_id;

REFRESH MATERIALIZED VIEW dept_stats;
```

Advanced Features:

- **Updatable Views**: (PostgreSQL/MySQL) With rules/triggers.
- **CONCURRENTLY**: (PostgreSQL) Refresh without locking.
- **WITH CHECK OPTION**: Enforce view conditions on updates.

Edge Cases:

- **Performance**: Views may hide inefficient queries.

- **Dependencies**: Dropping tables breaks views.
- **Storage**: Materialized views consume disk.

Cross-Connections:

- **CTEs**: Temporary views.
- **Indexes**: Index materialized views.
- **Security**: Restrict access via views.

JSON and NoSQL Features

Handle semi-structured data:

```
-- MySQL
CREATE TABLE products (
    id INTEGER PRIMARY KEY,
    details JSON
);

INSERT INTO products (id, details)
VALUES (1, '{"name": "Laptop", "price": 999.99}');

SELECT details->>'name' AS name FROM products;

-- PostgreSQL
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    details JSONB
);

INSERT INTO products (details)
VALUES ('{"name": "Laptop", "price": 999.99}'::JSONB);

SELECT details->>'name' AS name FROM products WHERE details @> '{"price":
999.99}'::JSONB;
```

Advanced Features:

- **JSONB**: (PostgreSQL) Binary JSON with indexing.
- **JSON Functions**: `JSON_EXTRACT` (MySQL), `json_agg` (PostgreSQL).
- **GIN Indexes**: Optimize JSON queries.

Edge Cases:

- **Performance**: JSON queries slower than columns.
- **Validation**: No schema enforcement.
- **Vendor Support**: SQLite has limited JSON.

Cross-Connections:

- **Indexes**: Optimize JSON searches.

- **Views**: Present JSON as columns.
- **NoSQL**: Bridge to MongoDB-like systems.

Security in SQL

User Management

Control access with roles and privileges:

```
-- PostgreSQL
CREATE ROLE app_user WITH LOGIN PASSWORD 'securepass';
GRANT SELECT, INSERT ON employees TO app_user;
REVOKE DELETE ON employees FROM app_user;

-- MySQL
CREATE USER 'app_user'@'localhost' IDENTIFIED BY 'securepass';
GRANT SELECT, INSERT ON myapp.employees TO 'app_user'@'localhost';
FLUSH PRIVILEGES;
```

Advanced Features:

- **Row-Level Security**: (PostgreSQL) Restrict rows by user.
- **Roles**: Group privileges for easier management.
- **Audit Logs**: Track access (vendor-specific).

Edge Cases:

- **Privilege Escalation**: Avoid granting `ALL`.
- **Connection Limits**: Restrict concurrent logins.
- **Password Management**: Use strong, rotated passwords.

Cross-Connections:

- **Views**: Limit data exposure.
- **Triggers**: Log access attempts.
- **Transactions**: Secure operations.

Preventing SQL Injection

Use parameterized queries:

```
-- Prepared statement (pseudo-SQL, syntax varies)
PREPARE stmt FROM 'SELECT * FROM employees WHERE name = ?';
EXECUTE stmt USING 'Alice';
DEALLOCATE PREPARE stmt;
```

Best Practices:

- **Never Concatenate**: Avoid `WHERE name = '$input'`.
- **ORMs**: Use libraries like Doctrine, Sequelize.

- **Input Validation**: Sanitize before querying.

Edge Cases:

- **Dynamic SQL**: Use whitelists for table/column names.
- **Stored Procedures**: Still vulnerable if inputs concatenated.
- **Legacy Code**: Audit for injection risks.

Cross-Connections:

- **Security**: User privileges reduce attack surface.
- **Performance**: Parameterized queries cache plans.
- **Application Code**: Integrate with PDO, JDBC.

Data Encryption

Protect sensitive data:

```
-- PostgreSQL (pgcrypto)
CREATE EXTENSION IF NOT EXISTS pgcrypto;
INSERT INTO employees (id, name, ssn)
VALUES (1, 'Alice', pgp_sym_encrypt('123-45-6789', 'secretkey'));

SELECT pgp_sym_decrypt(ssn, 'secretkey') AS ssn FROM employees;

-- MySQL
INSERT INTO employees (id, name, ssn)
VALUES (1, 'Alice', AES_ENCRYPT('123-45-6789', 'secretkey'));

SELECT AES_DECRYPT(ssn, 'secretkey') AS ssn FROM employees;
```

Advanced Features:

- **Transparent Data Encryption**: (SQL Server/Oracle) Encrypts at rest.
- **Column Encryption**: Encrypt specific columns.
- **Key Management**: Store keys securely (e.g., AWS KMS).

Edge Cases:

- **Performance**: Encryption adds overhead.
- **Key Loss**: Data becomes unrecoverable.
- **Indexing**: Encrypted columns can’t be indexed directly.

Cross-Connections:

- **Security**: Complements user access.
- **JSON**: Encrypt JSON fields.
- **Backups**: Ensure encrypted data in backups.

Sample Project: Task Management Database

This project creates a task management database with tables, constraints, triggers, views, and complex queries, demonstrating SQL's power.

Schema:

```
-- Create tables
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTO_INCREMENT, -- MySQL
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE tasks (
    id INTEGER PRIMARY KEY AUTO_INCREMENT, -- MySQL
    user_id INTEGER NOT NULL,
    description TEXT NOT NULL,
    status VARCHAR(20) DEFAULT 'pending' CHECK (status IN ('pending', 'completed',
'failed')),
    priority INTEGER DEFAULT 0 CHECK (priority BETWEEN 0 AND 10),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);

CREATE TABLE task_audit (
    audit_id INTEGER PRIMARY KEY AUTO_INCREMENT, -- MySQL
    task_id INTEGER NOT NULL,
    old_status VARCHAR(20),
    new_status VARCHAR(20),
    changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (task_id) REFERENCES tasks(id) ON DELETE CASCADE
);

-- PostgreSQL variants
-- Replace AUTO_INCREMENT with SERIAL
-- Add: CREATE EXTENSION IF NOT EXISTS pgcrypto;
-- Encrypt email: email VARCHAR(100) UNIQUE NOT NULL DEFAULT pgp_sym_encrypt(email,
'secretkey')
```

Indexes:

```
CREATE INDEX idx_tasks_user_id ON tasks (user_id);
CREATE INDEX idx_tasks_status ON tasks (status);
CREATE UNIQUE INDEX idx_users_email ON users (email);
```

Trigger (MySQL):

```
DELIMITER //
CREATE TRIGGER task_status_trigger
AFTER UPDATE ON tasks
FOR EACH ROW
BEGIN
    IF OLD.status != NEW.status THEN
        INSERT INTO task_audit (task_id, old_status, new_status)
```

```
        VALUES (NEW.id, OLD.status, NEW.status);
    END IF;
END //
DELIMITER ;
```

PostgreSQL Trigger:

```
CREATE FUNCTION log_task_status()
RETURNS TRIGGER
AS $$
BEGIN
    IF OLD.status != NEW.status THEN
        INSERT INTO task_audit (task_id, old_status, new_status)
        VALUES (NEW.id, OLD.status, NEW.status);
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER task_status_trigger
AFTER UPDATE ON tasks
FOR EACH ROW
EXECUTE FUNCTION log_task_status();
```

View:

```
CREATE VIEW user_task_summary AS
SELECT
    u.username,
    COUNT(t.id) AS total_tasks,
    COUNT(CASE WHEN t.status = 'completed' THEN 1 END) AS completed_tasks,
    AVG(t.priority) AS avg_priority
FROM users u
LEFT JOIN tasks t ON u.id = t.user_id
GROUP BY u.id, u.username;
```

Stored Procedure (MySQL):

```
DELIMITER //
CREATE PROCEDURE complete_task(IN task_id INT, OUT success BOOLEAN)
BEGIN
    DECLARE task_count INT;
    START TRANSACTION;
    SELECT COUNT(*) INTO task_count FROM tasks WHERE id = task_id AND status =
'pending';
    IF task_count = 1 THEN
        UPDATE tasks
        SET status = 'completed', updated_at = CURRENT_TIMESTAMP
        WHERE id = task_id;
        SET success = TRUE;
        COMMIT;
    ELSE
        SET success = FALSE;
        ROLLBACK;
    END IF;
END //
```

```
DELIMITER ;

CALL complete_task(1, @success);
SELECT @success;
```

PostgreSQL Procedure:

```
CREATE PROCEDURE complete_task(task_id INT, INOUT success BOOLEAN)
LANGUAGE plpgsql
AS $$
BEGIN
    IF EXISTS (SELECT 1 FROM tasks WHERE id = task_id AND status = 'pending') THEN
        UPDATE tasks
        SET status = 'completed', updated_at = CURRENT_TIMESTAMP
        WHERE id = task_id;
        success := TRUE;
    ELSE
        success := FALSE;
    END IF;
END;
$$;

CALL complete_task(1, TRUE);
```

Sample Queries:

```
-- Insert data
INSERT INTO users (username, email)
VALUES ('alice', 'alice@example.com'), ('bob', 'bob@example.com');

INSERT INTO tasks (user_id, description, status, priority)
VALUES
    (1, 'Finish report', 'pending', 8),
    (1, 'Call client', 'completed', 5),
    (2, 'Deploy app', 'pending', 9);

-- Complex query: Find users with high-priority pending tasks
WITH high_priority AS (
    SELECT user_id, COUNT(*) AS high_priority_count
    FROM tasks
    WHERE status = 'pending' AND priority >= 8
    GROUP BY user_id
)
SELECT u.username, h.high_priority_count,
       RANK() OVER (ORDER BY h.high_priority_count DESC) AS priority_rank
FROM users u
JOIN high_priority h ON u.id = h.user_id;

-- Query view
SELECT * FROM user_task_summary;

-- Audit log
SELECT t.description, a.old_status, a.new_status, a.changed_at
FROM tasks t
JOIN task_audit a ON t.id = a.task_id;
```

Features Demonstrated:

- **Schema Design**: Normalized tables with constraints.
- **Data Manipulation**: Inserts, updates, deletes.
- **Advanced Querying**: CTEs, window functions, joins.
- **Triggers**: Audit status changes.
- **Views**: Summarize data.
- **Stored Procedures**: Encapsulate logic.
- **Indexes**: Optimize queries.
- **Security**: Prepared statements, user privileges.
- **Transactions**: Ensure consistency.
- **Portability**: MySQL and PostgreSQL variants.

Running the Project:

1. Install MySQL or PostgreSQL.
2. Create database: `CREATE DATABASE task_manager;`.
3. Run schema and trigger SQL.
4. Insert sample data.
5. Execute queries via CLI (e.g., `mysql`, `psql`) or GUI (e.g., DBeaver).

Sample CLI (MySQL):

```
mysql -u root -p task_manager < schema.sql
mysql -u root -p task_manager
mysql> CALL complete_task(1, @success);
mysql> SELECT * FROM user_task_summary;
```

Edge Cases Handled:

- **NULL Handling**: Constraints and `COALESCE` in queries.
- **Concurrency**: Transactions prevent race conditions.
- **Performance**: Indexes on `user_id`, `status`, `email`.
- **Security**: No injection risks; triggers log changes.
- **Portability**: Vendor-specific syntax separated.

Resources

- **Official Docs**:
  - MySQL Manual,
  - PostgreSQL Docs,

    - SQLite Docs,
    - SQL Server Docs.
- **Standards**: SQL:2016 (overview via academic resources).
- **Tutorials**:
    - SQLZoo,
    - Mode Analytics SQL Tutorial,
    - PostgreSQL Tutorial.
- **Community**:
    - Stack Overflow,
    - Reddit r/SQL,
    - DBA Stack Exchange.
- **Tools**:
    - DBeaver,
    - pgAdmin,
    - SQL Fiddle.
- **Books**:
    - *SQL in a Nutshell* (O'Reilly),
    - *SQL Performance Explained* (Markus Winand),
    - *Joe Celko's SQL for Smarties* (Morgan Kaufmann).
- **Performance**:
    - Use The Index, Luke,
    - PostgreSQL EXPLAIN Visualizer.
- **Security**: OWASP SQL Injection.

This guide and sample project provide a comprehensive foundation for mastering SQL, from basic queries to advanced analytics, with practical applications and deep insights into relational database management.