SQLite Cheatsheet

Essential operations for database management and SQL queries

This cheatsheet provides a quick reference to fundamental SQLite operations, syntax, and advanced features, ideal for both beginners and experienced developers for efficient database management and data manipulation.

Database Setup
Create and connect to databases

Table Operations
Create, modify, and manage tables

Data Manipulation
Insert, update, and delete records

Data Querying
Select and filter data efficiently

Advanced Features
Joins, indexes, and optimization

Database Creation & Connection

Create Database: `sqlite3 database.db`

Create a new SQLite database file.

```
# Create or open a database
sqlite3 mydata.db
# Create in-memory database (temporary)
sqlite3 :memory:
# Create database with command
.open mydata.db
```

Database Info: `.databases`

List all attached databases and their files.

```
# Show all databases
.databases
# Show schema of all tables
.schema
# Show table list
.tables
```

Exit SQLite: `.exit` or `.quit`

Close the SQLite command-line interface.

```
# Exit SQLite
.exit
# Alternative exit command
.quit
```

Attach Database: `ATTACH`

Attach additional database files to current session.

```
-- Attach another database
ATTACH DATABASE 'backup.db' AS backup;
-- Query from attached database
SELECT * FROM backup.users;
-- Detach database
DETACH DATABASE backup;
```

Backup Database: `.backup`

Create a backup of the current database.

```
# Backup to file
.backup backup.db
# Restore from backup
.restore backup.db
```

Import/Export: `.dump` / `.read`

Export database to SQL script or import from file.

```
# Export to SQL file
.output backup.sql
.dump
# Import SQL script
.read backup.sql
```

Table Creation & Schema

Define database structure and create tables with proper data types and constraints.

01

Create Table: `CREATE TABLE`

Create a new table in the database with columns and constraints.

```
-- Basic table creation
CREATE TABLE users (
  id INTEGER PRIMARY KEY
AUTOINCREMENT,
  name TEXT NOT NULL,
  email TEXT UNIQUE,
  age INTEGER,
  created_date DATE DEFAULT
CURRENT_TIMESTAMP
);

-- Table with foreign key
CREATE TABLE orders (
  id INTEGER PRIMARY KEY,
  user_id INTEGER,
  amount REAL,
  FOREIGN KEY (user_id)
REFERENCES users(id)
);
```

02

Data Types: `INTEGER`, `TEXT`, `REAL`, `BLOB`

SQLite uses dynamic typing with storage classes for flexible data storage.

```
-- Common data types
CREATE TABLE products (
  id INTEGER,         -- Whole
numbers
  name TEXT,          -- Text strings
  price REAL,         -- Floating point
numbers
  image BLOB,         -- Binary data
  active BOOLEAN,     -- Boolean
(stored as INTEGER)
  created_at DATETIME  -- Date and
time
);
```

03

Constraints: `PRIMARY KEY`, `NOT NULL`, `UNIQUE`

Define constraints to enforce data integrity and table relationships.

```
CREATE TABLE employees (
  id INTEGER PRIMARY KEY
AUTOINCREMENT,
  email TEXT NOT NULL UNIQUE,
  department TEXT NOT NULL,
  salary REAL CHECK(salary > 0),
  manager_id INTEGER REFERENCES
employees(id)
);
```

Data Insertion & Modification

Insert Data: `INSERT INTO`

Add new records to tables with single or multiple rows.

```
-- Insert single record
INSERT INTO users (name, email, age)
VALUES ('John Doe', 'john@email.com', 30);

-- Insert multiple records
INSERT INTO users (name, email, age) VALUES
  ('Jane Smith', 'jane@email.com', 25),
  ('Bob Wilson', 'bob@email.com', 35);

-- Insert with all columns
INSERT INTO users VALUES
  (NULL, 'Alice Brown', 'alice@email.com', 28,
datetime('now'));
```

Update Data: `UPDATE SET`

Modify existing records based on conditions.

```
-- Update single column
UPDATE users SET age = 31 WHERE name = 'John Doe';

-- Update multiple columns
UPDATE users SET
  email = 'newemail@example.com',
  age = age + 1
WHERE id = 1;

-- Update with subquery
UPDATE products SET price = price * 1.1
WHERE category = 'Electronics';
```

Delete Data: `DELETE FROM`

Remove records from tables based on specified conditions.

```
-- Delete specific records
DELETE FROM users WHERE age < 18;

-- Delete all records (keep table structure)
DELETE FROM users;

-- Delete with subquery
DELETE FROM orders
WHERE user_id IN (SELECT id FROM users WHERE active =
0);
```

Upsert: `INSERT OR REPLACE`

Insert new records or update existing ones based on conflicts.

```
-- Insert or replace on conflict
INSERT OR REPLACE INTO users (id, name, email)
VALUES (1, 'Updated Name', 'updated@email.com');

-- Insert or ignore duplicates
INSERT OR IGNORE INTO users (name, email)
VALUES ('Duplicate', 'existing@email.com');
```

Data Querying & Selection

Basic Queries: `SELECT`

Query data from tables using SELECT statement with various options.

```
-- Select all columns
SELECT * FROM users;

-- Select specific columns
SELECT name, email FROM users;

-- Select with alias
SELECT name AS full_name, age AS years_old FROM
users;

-- Select unique values
SELECT DISTINCT department FROM employees;
```

Filtering: `WHERE`

Filter rows using various conditions and comparison operators.

```
-- Simple conditions
SELECT * FROM users WHERE age > 25;
SELECT * FROM users WHERE name = 'John Doe';

-- Multiple conditions
SELECT * FROM users WHERE age > 18 AND age < 65;
SELECT * FROM users WHERE department = 'IT' OR salary
> 50000;

-- Pattern matching
SELECT * FROM users WHERE email LIKE '%@gmail.com';
SELECT * FROM users WHERE name GLOB 'J*';
```

Sorting & Limiting: `ORDER BY` / `LIMIT`

Sort results and limit the number of rows returned for better data management.

```
-- Sort ascending (default)
SELECT * FROM users ORDER BY age;

-- Sort descending
SELECT * FROM users ORDER BY age DESC;

-- Multiple sort columns
SELECT * FROM users ORDER BY department, salary
DESC;

-- Limit results
SELECT * FROM users LIMIT 10;

-- Limit with offset (pagination)
SELECT * FROM users LIMIT 10 OFFSET 20;
```

Aggregate Functions: `COUNT`, `SUM`, `AVG`

Perform calculations on groups of rows for statistical analysis.

```
-- Count records
SELECT COUNT(*) FROM users;
SELECT COUNT(DISTINCT department) FROM employees;

-- Sum and average
SELECT SUM(salary), AVG(salary) FROM employees;

-- Min and max values
SELECT MIN(age), MAX(age) FROM users;
```

Advanced Querying

Grouping: `GROUP BY` / `HAVING`

Group rows by specified criteria and filter groups for summary reporting.

```
-- Group by single column
SELECT department, COUNT(*)
FROM employees
GROUP BY department;

-- Group by multiple columns
SELECT department, job_title, AVG(salary)
FROM employees
GROUP BY department, job_title;

-- Filter groups with HAVING
SELECT department, AVG(salary) as avg_salary
FROM employees
GROUP BY department
HAVING avg_salary > 60000;
```

Subqueries

Use nested queries for complex data retrieval and conditional logic.

```
-- Subquery in WHERE clause
SELECT name FROM users
WHERE age > (SELECT AVG(age) FROM users);

-- Subquery in FROM clause
SELECT dept, avg_salary FROM (
  SELECT department as dept, AVG(salary) as avg_salary
  FROM employees
  GROUP BY department
) WHERE avg_salary > 50000;

-- EXISTS subquery
SELECT * FROM users u
WHERE EXISTS (
  SELECT 1 FROM orders o WHERE o.user_id = u.id
);
```

Joins: `INNER`, `LEFT`, `RIGHT`

Combine data from multiple tables using various join types for relational queries.

```
-- Inner join
SELECT u.name, o.amount
FROM users u
INNER JOIN orders o ON u.id = o.user_id;

-- Left join (show all users)
SELECT u.name, COUNT(o.id) as order_count
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
GROUP BY u.id, u.name;

-- Self join
SELECT e1.name as employee, e2.name as manager
FROM employees e1
LEFT JOIN employees e2 ON e1.manager_id = e2.id;
```

Set Operations: `UNION` / `INTERSECT`

Combine results from multiple queries using set operations.

```
-- Union (combine results)
SELECT name FROM customers
UNION
SELECT name FROM suppliers;

-- Intersect (common results)
SELECT email FROM users
INTERSECT
SELECT email FROM newsletter_subscribers;

-- Except (difference)
SELECT email FROM users
EXCEPT
SELECT email FROM unsubscribed;
```

Indexes & Performance

Optimize query performance and database efficiency with proper indexing strategies.

Create Indexes: `CREATE INDEX`

Create indexes on columns to speed up queries and improve performance.

```
-- Single column index
CREATE INDEX idx_user_email ON users(email);

-- Multi-column index
CREATE INDEX idx_order_user_date ON orders(user_id,
order_date);

-- Unique index
CREATE UNIQUE INDEX idx_product_sku ON
products(sku);

-- Partial index (with condition)
CREATE INDEX idx_active_users ON users(name) WHERE
active = 1;
```

Query Analysis: `EXPLAIN QUERY PLAN`

Analyze query execution plans to identify performance bottlenecks.

```
-- Analyze query performance
EXPLAIN QUERY PLAN SELECT * FROM users WHERE
email = 'test@example.com';

-- Check if index is being used
EXPLAIN QUERY PLAN SELECT * FROM orders WHERE
user_id = 123;
```

Database Optimization: `VACUUM` / `ANALYZE`

Optimize database files and update statistics for better performance.

```
-- Rebuild database to reclaim space
VACUUM;

-- Update index statistics
ANALYZE;

-- Check database integrity
PRAGMA integrity_check;
```

Performance Settings: `PRAGMA`

Configure SQLite settings for optimal performance and behavior.

```
-- Set journal mode for better performance
PRAGMA journal_mode = WAL;

-- Set synchronous mode
PRAGMA synchronous = NORMAL;

-- Enable foreign key constraints
PRAGMA foreign_keys = ON;

-- Set cache size (in pages)
PRAGMA cache_size = 10000;
```

Views & Triggers

Views: `CREATE VIEW`

Create virtual tables that represent stored queries for reusable data access.

```
-- Create a simple view
CREATE VIEW active_users AS
SELECT id, name, email
FROM users
WHERE active = 1;

-- Complex view with joins
CREATE VIEW order_summary AS
SELECT
  u.name,
  COUNT(o.id) as total_orders,
  SUM(o.amount) as total_spent
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
GROUP BY u.id, u.name;
```

Using Views

Query views like regular tables for simplified data access.

```
-- Query a view
SELECT * FROM active_users WHERE name LIKE 'J%';

-- Drop a view
DROP VIEW IF EXISTS order_summary;
```

Triggers: `CREATE TRIGGER`

Automatically execute code in response to database events.

```
-- Trigger on INSERT
CREATE TRIGGER update_user_count
AFTER INSERT ON users
BEGIN
  UPDATE stats SET user_count = user_count + 1;
END;

-- Trigger on UPDATE
CREATE TRIGGER log_salary_changes
AFTER UPDATE OF salary ON employees
BEGIN
  INSERT INTO audit_log (table_name, action, old_value,
new_value)
  VALUES ('employees', 'salary_update', OLD.salary,
NEW.salary);
END;

-- Drop trigger
DROP TRIGGER IF EXISTS update_user_count;
```

Data Types & Functions

Work with different data types and utilize built-in SQLite functions for data manipulation.

Date & Time Functions

Handle date and time operations with SQLite's built-in functions.

```
-- Current date/time
SELECT datetime('now');
SELECT date('now');
SELECT time('now');

-- Date arithmetic
SELECT date('now', '+1 day');
SELECT datetime('now', '-1 hour');
SELECT date('now', 'start of month');

-- Format dates
SELECT strftime('%Y-%m-%d %H:%M', 'now');
SELECT strftime('%w', 'now'); -- day of week
```

String Functions

Manipulate text data with various string operations.

```
-- String manipulation
SELECT upper(name) FROM users;
SELECT lower(email) FROM users;
SELECT length(name) FROM users;
SELECT substr(name, 1, 3) FROM users;

-- String concatenation
SELECT name || ' - ' || email as display FROM users;
SELECT printf('%s (%d)', name, age) FROM users;

-- String replacement
SELECT replace(phone, '-', '') FROM users;
```

Numeric Functions

Perform mathematical operations and calculations.

```
-- Mathematical functions
SELECT abs(-15);
SELECT round(price, 2) FROM products;
SELECT random(); -- random number

-- Aggregation with math
SELECT department, round(AVG(salary), 2) as avg_salary
FROM employees
GROUP BY department;
```

Conditional Logic: `CASE`

Implement conditional logic within SQL queries.

```
-- Simple CASE statement
SELECT name,
  CASE
    WHEN age < 18 THEN 'Minor'
    WHEN age < 65 THEN 'Adult'
    ELSE 'Senior'
  END as age_category
FROM users;

-- CASE in WHERE clause
SELECT * FROM products
WHERE CASE WHEN category = 'Electronics' THEN price <
1000
      ELSE price < 100 END;
```

Transactions & Concurrency

Handle transactions in SQLite to ensure data integrity and manage concurrent access.

Transaction Control

SQLite transactions are fully ACID-compliant for reliable data operations.

```
-- Basic transaction
BEGIN TRANSACTION;
INSERT INTO users (name, email) VALUES ('Test User',
'test@example.com');
UPDATE users SET age = 25 WHERE name = 'Test User';
COMMIT;

-- Transaction with rollback
BEGIN;
DELETE FROM orders WHERE amount < 10;
-- Check results, rollback if needed
ROLLBACK;

-- Savepoints for nested transactions
BEGIN;
SAVEPOINT sp1;
INSERT INTO products (name) VALUES ('Product A');
ROLLBACK TO sp1;
COMMIT;
```

Locking & Concurrency

Manage database locks and concurrent access for data integrity.

```
-- Check lock status
PRAGMA locking_mode;

-- Set WAL mode for better concurrency
PRAGMA journal_mode = WAL;

-- Busy timeout for waiting on locks
PRAGMA busy_timeout = 5000;

-- Check current database connections
.databases
```

SQLite Command Line Tools

Utilize SQLite's command-line interface for database administration and development tasks.

Database Commands: `.help`

Access SQLite command-line help and documentation for available dot commands.

```
# Show all available commands
.help
# Show current settings
.show
# Set output format
.mode csv
.headers on
```

Import/Export: `.import` / `.export`

Transfer data between SQLite and external files in various formats.

```
# Import CSV file
.mode csv
.import data.csv users
# Export to CSV
.headers on
.mode csv
.output users.csv
SELECT * FROM users;
```

Schema Management: `.schema` / `.tables`

Examine database structure and table definitions for development and debugging.

```
# Show all tables
.tables
# Show schema for specific
table
.schema users
# Show all schemas
.schema
# Show table info
.mode column
.headers on
PRAGMA table_info(users);
```

Configuration & Settings

Configure SQLite behavior and optimize settings for specific use cases and environments.

Database Settings: `PRAGMA`

Control SQLite's behavior through pragma statements for optimization and configuration.

```
-- Database information
PRAGMA database_list;
PRAGMA table_info(users);
PRAGMA foreign_key_list(orders);

-- Performance settings
PRAGMA cache_size = 10000;
PRAGMA temp_store = memory;
PRAGMA mmap_size = 268435456;
```

Security Settings

Configure security-related database options and constraints.

```
-- Enable foreign key constraints
PRAGMA foreign_keys = ON;

-- Set secure delete mode
PRAGMA secure_delete = ON;

-- Check constraints
PRAGMA foreign_key_check;
```

Output Formatting: `.mode`

Control how query results are displayed in the command-line interface.

```
# Different output modes
.mode csv      # Comma-separated values
.mode column   # Aligned columns
.mode html     # HTML table format
.mode json     # JSON format
.mode list     # List format
.mode table    # Table format (default)

# Set column width
.width 10 15 20
```

File Operations

Manage database files and perform file-related operations.

```
# Save output to file
.output results.txt
SELECT * FROM users;
.output stdout

# Read SQL from file
.read script.sql

# Change database file
.open another_database.db
```

Installation & Setup

Get started with SQLite by installing and configuring the database engine for your development environment.

Download & Install

Download SQLite tools and set up the command-line interface for your operating system.

```
# Download from sqlite.org
# For Windows: sqlite-tools-win32-x86-*.zip
# For Linux/Mac: Use package manager

# Ubuntu/Debian
sudo apt-get install sqlite3

# macOS with Homebrew
brew install sqlite

# Verify installation
sqlite3 --version
```

Creating Your First Database

Create SQLite database files and start working with data using simple commands.

```
# Create new database
sqlite3 myapp.db

# Create table and add data
CREATE TABLE users (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  email TEXT UNIQUE
);

INSERT INTO users (name, email)
VALUES ('John Doe', 'john@example.com');
```

Programming Language Integration

Use SQLite with various programming languages through built-in or third-party libraries.

```
# Python (built-in sqlite3 module)
import sqlite3
conn = sqlite3.connect('mydb.db')
cursor = conn.cursor()
cursor.execute('SELECT * FROM users')

# Node.js (requires sqlite3 package)
const sqlite3 = require('sqlite3');
const db = new sqlite3.Database('mydb.db');
db.all('SELECT * FROM users', (err, rows) => {
  console.log(rows);
});

# PHP (built-in PDO SQLite)
$pdo = new PDO('sqlite:mydb.db');
$stmt = $pdo->query('SELECT * FROM users');
```

Reference: This cheatsheet covers essential SQLite commands and best practices for efficient database development and management in various applications and environments.