

Unit – 1 Introduction to SQLite

Que) What is SQLite?

Ans: SQLite is embedded relational database management system.

- It is **self-contained, server less, zero configuration** and **transactional** SQL database engine.
- SQLite is free to use for any purpose commercial or private.
- In other words, "SQLite is an open source, zero-configuration, self-contained, stand alone, transaction relational database engine designed to be embedded into an application".
- SQLite is different from other SQL databases because unlike most other SQL databases, SQLite does not have a separate server process. It reads and writes directly to ordinary disk files.

Que) Why Use SQLite?

Ans: SQLite is **open-source** software. The software does not require any license after installation.

- SQLite is **server less** as it doesn't need a different server process or system to operate.
- SQLite facilitates you to work on multiple databases on the same session simultaneously, thus making it flexible.
- SQLite is a **cross-platform** DBMS that can run on all platforms, including macOS, Windows, etc.
- SQLite doesn't require any configuration. It needs no setup or administration.

Que) When to Use SQLite?

Ans: It can manage low to medium-traffic HTTP requests.

- SQLite is used to develop embedded software for devices like televisions, cell phones, cameras, etc.
- SQLite can change files into smaller size archives with lesser metadata.
- SQLite is used as a temporary dataset to get processed with some data within an application.

Que) SQLite History

Ans: SQLite was designed originally on **August 2000** by **D. Richard Hipp** for the purpose of no administration required for operating a program. In August SQLite 1.0 released with GNU database manager. It is named SQLite because it is very light weight (less than 500Kb size).

Que) What are Advantages and Limitation of SQLite?

SQLite Advantages:

1. **Lightweight:**

SQLite is a very light weighted database so, it is easy to use it as an embedded software with devices like televisions, mobile phones, cameras, etc.

2. **Better Performance:**

Reading and writing operations are very fast for SQLite database.

3. **No installation needed:**

SQLite is very easy to learn. You don't need to install and configuration it. Just download SQLite libraries in your computer and it is ready for creating the database.

4. **Reliable:**

It updates your content continuously so, little or no work is lost in a case of power failure or crash. SQLite is queries are smaller than equivalent procedural codes so, changes of bugs are minimal.

5. **Portable:**

SQLite is portable across all 32-bit and 64 bit OS. It can be used with all programming languages without any compatibility issue.

6. Accessible:

SQLite database is accessible through a wide variety of third-party tools.

7. Reduce cost and Complexity:

It reduces application cost because content can be accessed and updated using SQL queries.

SQLite Disadvantages (Limitations)

1. SQLite is used to handle low to medium traffic HTTP requests.
2. Database size is restricted to 2GB in most cases.

Que) Explain SQLite features. OR What are the features of SQLite?

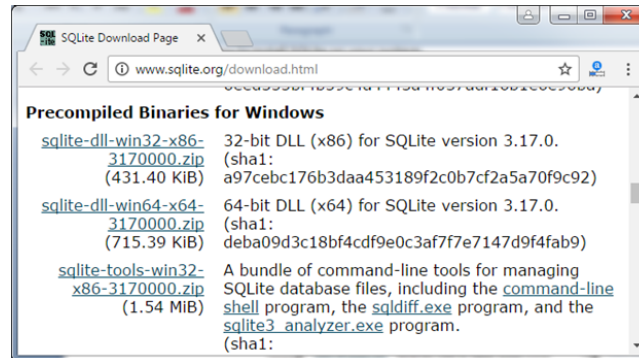
1. **SQLite is totally free:** It is open-source. So, no license is required to work with it.
2. **SQLite is server less:** It doesn't require a different server process or system to operate.
3. **SQLite is very flexible:** It facilitates you to work on multiple databases on the same session on the same time.
4. **Configuration not required:** SQLite doesn't required configuration. No setup or administration required.
5. **Storing data is easy:** SQLite provides an efficient way to store data.
6. **Provide large number of API's:** SQLite provides API for a large range of programming languages. For example: .NET languages, PHP, JAVA, Python.
7. **SQLite is written in ANSI-C** and provides simple and easy to use API's.

Que) Differences between SQL and SQLite

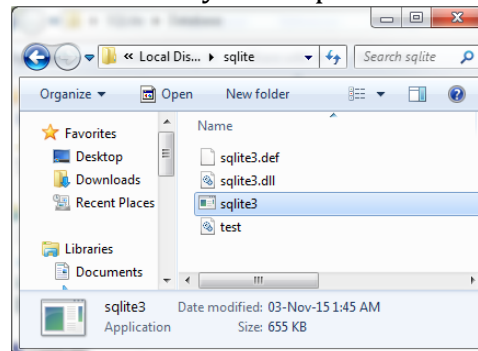
Aspect	SQL	SQLite
Definition	A language Structured Query Language) used to query and manage relational databases.	A database engine that implements SQL in a lightweight, server less form.
Architecture	Works in client-server model; requires a running database server.	Serverless ; directly reads/writes to disk files.
Deployment	Needs installation, configuration, and user management.	Zero-configuration; portable and easy to embed in apps.
Data Types	Strict typing (INTEGER, VARCHAR, etc.).	Dynamic typing; columns can store any type of value.
Use Cases	Large-scale enterprise systems, multi-user applications, high concurrency.	Mobile apps, browsers, IoT devices, small projects, prototyping.
Nature	Standardized by ANSI used across systems like MySQL, PostgreSQL, Oracle, SQL Server.	Embedded library written in C creates a single .db file for storage.
Examples	MySQL, PostgreSQL, OracleDB, SQL Server.	Android apps, iOS apps, Chrome, Firefox, embedded systems.

SQLite Installation

1. Go to SQLite official website download page <http://www.sqlite.org/download.html> And download precompiled binaries from Windows section.



2. Download the sqlite-dll and sqlite-shell zip file. Or sqlite-tools-win32-x86-3170000.zip file.
3. Create a folder named sqlite in C directory and expand these files.

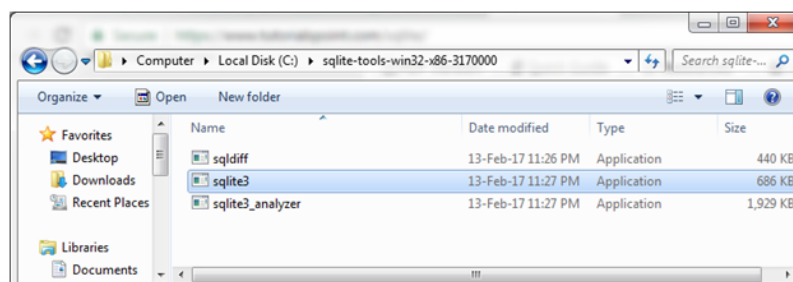


4. Open command prompt to set the path. Set your PATH environment variable and open sqlite3 command. It will look like this:

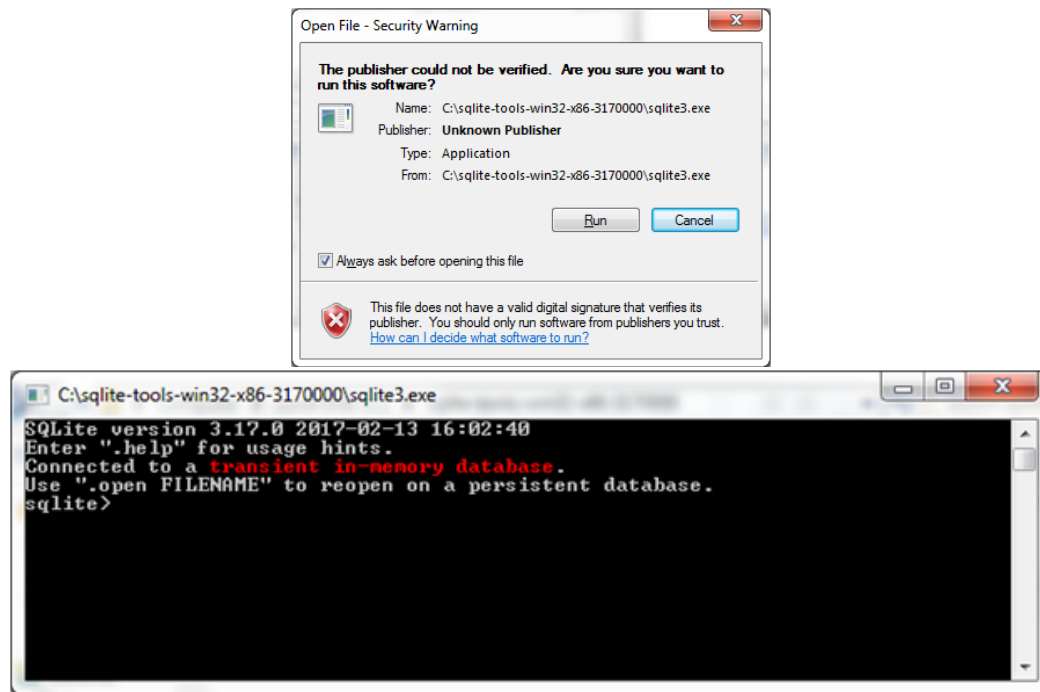


There is another way to execute CRUD operation in SQLite. In this method, there is no need to set a path.

1. Just download the SQLite precompiled Binary zip file.
2. Expand the zipped file, you will see a page like this:



3. Run the selected sqlite3 application:



You can execute the SQLite query here. But here, the data is temporary and once you shut down your computer, you will lose the records you have. Because, you cannot create, attach or detach a database here.

Que) Explain SQLite Data type.

Ans:

- SQLite data types are used to specify type of data of any object. In SQLite, each table in a database has a data type and name. The data type defines what type of data can be stored in a column of a table.
- Most databases use strong, **static** column type for tables. This means that the column can only hold a value that is compatible with columns defined type.
- SQLite uses a **dynamic typing** technique also known as **Manifest Typing**. This means it allows nearly any element of any row to hold almost any type of value.

SQLite Storage Classes

Storage Class	Description
NULL	It specifies that the value is a null value.
INTEGER	It specifies the value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.
REAL	It specifies the value is a floating point value, stored as an 8-byte IEEE floating point number.
text	It specifies the value is a text string, stored using the database encoding
BLOB	It specifies the value is a blob of data, stored exactly as it was input.

Que) SQLite Affinity Type

Ans: Type affinity determines the storage class. In SQLite, type affinity is used to store the values within a column, and the declared type of column determines the type affinity of a column. However, you still can store any type of data as you wish; these types are **recommended** but **not required**.

- In SQLite, columns don't have types or domains. While a column can have a declared type, internally it only has a type affinity. But the declared type and type affinity are two different things.
- A column's affinity is determined directly from its declared type. Therefore, when you declare a column in a table, the type you choose to declare it will ultimately determine that column's affinity.
- In SQLite, each table column must have one of **five** type affinities.

Data Type	Description
Text	A column with this affinity can only store TEXT, NULL or BLOB value. If you store INTEGER in this affinity, then it is converted to text value type.
Numeric	A column with a numeric affinity will store any of the five types. Values with integer and float types, along with NULL and BLOB types, are stored without conversion.
Integer	A column with an integer affinity works essentially the same as a numeric affinity. If you store float value, then it is converted to an integer type.
Float	A column with a floating-point affinity also works essentially the same as a numeric affinity. The only difference is if you store integer value then it is converted to floating value.
None	A column with none affinity has no preference over storage class. It is called as BLOB affinity.

Following table lists down various data type names which can be used while creating SQLite3 tables with the corresponding applied affinity.

Data Type	Affinity
• INT • INTEGER • TINYINT • SMALLINT • MEDIUMINT • BIGINT • UNSIGNED BIG INT • INT2 • INT8	INTEGER
• CHARACTER(20) • VARCHAR(255) • VARYING CHARACTER(255) • NCHAR(55) • NATIVE CHARACTER(70) • NVARCHAR(100) • TEXT • CLOB	TEXT
• BLOB • no datatype specified	NONE
• REAL • DOUBLE	REAL

• DOUBLE PRECISION • FLOAT	
• NUMERIC • DECIMAL(10,5) • BOOLEAN • DATE • DATETIME	NUMERIC

What is SQLite Manifest typing & type affinity?

Ans:

- Manifest typing means that a data type is a property of a value stored in a column, not the property of the column in which the value is stored. SQLite uses manifest typing to store values of any type in a column.
- Type affinity of a column is the recommended type for data stored in that column. Note that the data type is recommended, not required, therefore, a column can store any type of data.

SQLite Create Database

SQLite does not use CREATE DATABASE command to create the database.

Syntax: **Sqlite3 database-name.db**

Now we will see how to create a database using the SQLite command-line tool. For that open installed sqlite command-line tool directly or open it from the command prompt.

Open command prompt and navigate to the folder where your sqlite command-line tool installed. During installation, we created the **SQLite** folder in "C" directory and copied **sqlite3.exe** in it so now we are navigating to respective **SQLite** folder like as shown below.

```

Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\Admin-PC>cd c:\sqlite
c:\sqlite>
  
```

Once we navigated to SQLite installed folder now type the following command to create a database.

Sqlite3 db1.db

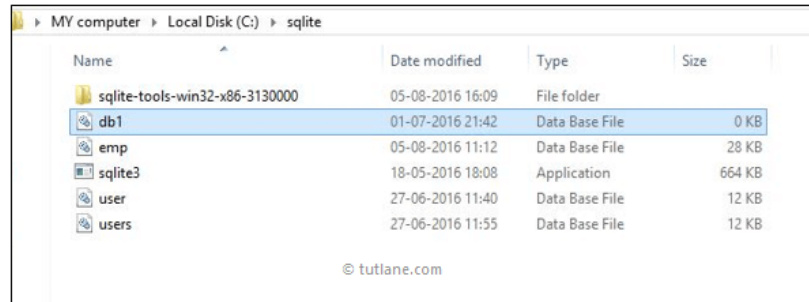
By the above command, SQLite creates a database named **db1**. If there is already a database called **db1** then it will open for you.

```

Command Prompt - sqlite3 db1.db

c:\sqlite>sqlite3 db1.db
SQLite version 3.13.0 2016-05-18 10:57:30
Enter ".help" for usage hints.
sqlite>
  
```

You can see the database that you have just created by redirecting to SQLite command-line location as follows.



You can see **db1.db** is created at **c:/sqlite** location. You can also ensure that the **db1** database is created or not by the following command.

.databases

From this command, you can see the list of databases created. You can also see **db1** on this list.



SQLite Attach Database

When we have multiple databases, we can use only one at a time. In SQLite, the ATTACH DATABASE statement is used to attach a particular database for the current connection.

In SQLite whenever we are connecting to a specific database, its name is main regardless of what's its filename. We can also work with a temporary database and a temporary table with the use of a temp database.

So, we can say that if you are connecting to a particular database, you can use two databases i.e. main and temp databases.

Syntax: ATTACH DATABASE 'dbfilename' AS alias_name;

The above SQLite statement is used to attach the database file "dbfilename" to the current connection with the name alias_name.

First, we connect to the emp.db database using the sqlite3 command as follows.

sqlite3 emp.db

We will fire .databases meta command to see the current connection.

sqlite> .databases

Here you can see that the emp.db database has given name main database. Now, we will try to attach some other database to the current connection using the ATTACH DATABASE command as follows.

```
ATTACH DATABASE 'C:\sqlite\db1.db' as myDB;
```

Now, we again fire .databases meta-command to see all active databases that can we use.

```
sqlite> .databases
```

Example:

```
CREATE TABLE myDB.dept_master(  
  dept_id INTEGER PRIMARY KEY AUTOINCREMENT, dept_name TEXT NOT NULL);  
INSERT INTO myDB.dept_master (dept_name) VALUES ('Engineering');  
  
SELECT * FROM mydb.dept_master;
```

SQLite Detach Database

In SQLite detach database command allows you to dissociate database from the current connection. The database which we are trying to detach is the one which we already opened with the ATTACH DATABASE command.

In case if the same database file has been attached multiple times in SQLite then the detach database statement will only detach mentioned name instance and the remaining attachments will continue with the current session. If we use SQLite detach command on the database which is an in-memory or temporary, the database will be destroyed and the content will be lost completely.

In SQLite, we cannot detach main or temp databases. The DETACH command will fail if issued inside a transaction.

Syntax: DETACH DATABASE 'database_alias_name';

Here database_alias_name is the alias name of the database that we give as a name for the database when using ATTACH DATABASE command.

Example: DETACH DATABASE empDB;

SQLite Create Table

Syntax:

```
CREATE TABLE table_name  
(  
  Column1 column_type [constraint],  
  Column2 column_type [constraint],  
  [...]  
);
```

Example:

```
CREATE TABLE emp_master  
(  
  ID INTEGER PRIMARY KEY AUTOINCREMENT,  
  NAME VARCHAR NOT NULL,  
  AGE NUMERIC,  
  SALARY NUMERIC  
);
```

SQLite Drop Table:**Syntax:**

```
DROP TABLE table_name;
```

Example:

```
DROP TABLE emp_master;
```

SQLite Insert Query:**Syntax:**

```
INSERT INTO TABLE_NAME [ (column1, column2, column3,...columnN)] VALUES (value1, value2, value3,...valueN);
```

OR

```
INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN);
```

Example:

```
INSERT INTO emp_master (ID,NAME,AGE, SALARY) VALUES (1, 'Paul', 32, 20000.00 );  
INSERT INTO emp_master (ID,NAME,AGE, SALARY) VALUES (2, 'Allen', 25, 15000.00 );
```

OR

```
INSERT INTO emp_master VALUES ('7', 'James', 24, 10000.00 );
```

SQLite Select Query:**Syntax:**

```
SELECT column1, column2, columnN FROM table_name;
```

Example:

```
SELECT * FROM table_name;
```

```
SELECT first_name, salary FROM emp_master;
```

Explain SQLite ACID Transactions:**Ans:**

- A transaction is a unit of work that is performed against a database.
- Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program.
- A transaction is the one or more changes to the database. For example, if you are creating, updating, or deleting a record from the table, then you are performing transaction on the table. It is important to control transactions to ensure data integrity and to handle database errors.
- If we are performing any operation on the table like INSERT, UPDATE or DELETE a record from the table then we will call it as the transaction is happening on the table and these transactions will follow ACID properties to ensure data integrity.
- In any DBMS the database transactions must follow ACID properties to produce consistent results and must be isolated from other database operations.

Atomicity:

- Atomicity means all the operations of transaction must be finished or rollback the complete transaction in case of any failures.
- During the transaction in case of any power failures, crash or some other error the complete transaction will be roll backed.
- An Atomic system must guarantee that atomicity will application in each and every situation including power failures, errors, and crashes.

Consistency:

- Consistent means it always keeps the data consistent and no database rules will be violated during any transaction on the database.
- The consistency ensures that any changes to the values in an instance are consistent with the changes to other values in the same instance.

Isolation:

- Isolation means one transaction cannot read the data from another transaction until that is completed. In case if two transactions executing concurrently only respective transaction results will appear to the client, not other clients transaction results.
- The main goal of the isolation process is to ensure the correct results for concurrent operations.

Durability:

- Durability means once the transaction committed successfully then it is guaranteed that all the changes committed permanently to the database and recorded to durable medium like hard disk regardless of power failure or other errors.
- In case, if a power fails or a program crash occur before the transaction committed then that changes will be roll backed.
- All these ACID (atomicity, consistency, isolation and durability) properties need to be satisfied by all the transactions which will happen in database.

Explain SQLite Transactions control (Begin, Commit, Rollback):**Ans:**

A transaction is a unit of work that is performed against a database. Following are Transaction Control

1. **BEGIN** – Start a transaction.
2. **COMMIT** – Save all changes permanently.
3. **ROLLBACK** – Cancel all changes since BEGIN (Undoes all changes since).

We can use these commands only when we are performing INSERT, UPDATE, and DELETE operations.

SQLite BEGIN Command

- The BEGIN command is used to start or open a transaction. Once an explicit transaction has been opened, it will remain open until it is committed or rolled back.

Syntax:-

```
BEGIN;  
or  
BEGIN [TRANSACTION];
```

Without BEGIN:

```
INSERT INTO users VALUES (1, 'Alice');  
INSERT INTO users VALUES (2, 'Bob');
```

This performs **2 separate transactions**.

With BEGIN:

```
BEGIN;  
INSERT INTO users VALUES (1, 'Alice');  
INSERT INTO users VALUES (2, 'Bob');  
COMMIT;
```

This performs **1 transaction**, which is much faster for bulk operations.

SQLite COMMIT Command

- COMMIT is used to permanently save the changes made during a transaction.
- COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.

Syntax :-

```
COMMIT;  
or  
END [TRANSACTION];
```

Example:

```
BEGIN;  
INSERT INTO users (id, name)VALUES (1, 'Alice');  
COMMIT;
```

After COMMIT, the new row is permanently stored in the database.

What happens without COMMIT?**Ans:**

```
BEGIN;  
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
```

At this point:

- The change exists only within the current transaction.
- If you run: ROLLBACK; The update is undone.
- If instead you run: COMMIT;

SQLite ROLLBACK Command

By using ROLLBACK command we can cancel the transaction and roll back all the proposed changes. Following is the syntax of the ROLLBACK command.

Syntax:

```
ROLLBACK  
or  
ROLLBACK [TRANSACTION]
```

Here also TRANSACTION keyword is optional.

Why is COMMIT needed?

- To save changes permanently.
- To end the transaction.
- To ensure all operations in the transaction succeed together.
- To release any locks held by the transaction.

Transaction flow

```
BEGIN; -- Start transaction  
INSERT ...  
UPDATE ...
```

```
DELETE ...
COMMIT;  -- Save everything permanently
```

Or, if something goes wrong:

```
BEGIN;
INSERT ...
UPDATE ...
ROLLBACK;  -- Undo all changes
```

SQLite Transaction Examples

To use transaction commands first we need to create a table called **emp_master** and insert some data using the following queries.

```
CREATE TABLE emp_master (emp_id INTEGER PRIMARY KEY AUTOINCREMENT,
first_name TEXT, last_name TEXT, salary NUMERIC, dept_id INTEGER);
```

```
INSERT INTO emp_master
values (1,'Honey','Patel', 10100,1),
(2,'Shweta','Jariwala', 19300,2),
(3,'Vinay','Jariwala', 35100,3),
(4,'Jagruti','Viras', 9500,2),
(5,'Shweta','Rana',12000,3),
(6,'sonal','Menpara', 13000,1),
(7,'Yamini','Patel', 10000,2),
(8,'Khyati','Shah', 50000,3),
(9,'Shwets','Jariwala',19400,2);
```

```
sqlite> SELECT * FROM emp_master;
```

emp_id	first_name	last_name	salary	dept_id
1	Honey	Patel	10100	1
2	Shweta	Jariwala	19300	2
3	Vinay	Jariwala	35100	3
4	Jagruti	Viras	9500	2
5	Shweta	Rana	12000	3
6	Sonal	Menpara	13000	1
7	Yamini	Patel	10000	2
8	Khyati	Shah	50000	3
9	Shwets	Jariwala	19400	2

```
BEGIN;
DELETE FROM emp_master WHERE dept_id=3;
ROLLBACK;
```

```
sqlite> SELECT * FROM emp_master;
```

emp_id	first_name	last_name	salary	dept_id
1	Honey	Patel	10100	1
2	Shweta	Jariwala	19300	2
3	Vinay	Jariwala	35100	3

4	Jagruti	Viras	9500	2
5	Shweta	Rana	12000	3
6	Sonal	Menpara	13000	1
7	Yamini	Patel	10000	2
8	Khyati	Shah	50000	3
9	Shwets	Jariwala	19400	2

```
BEGIN;
DELETE FROM emp_master WHERE emp_id=1;
COMMIT;
```

```
sqlite> SELECT * FROM emp_master;
```

emp_id	first_name	last_name	salary	dept_id
2	Shweta	Jariwala	19300	2
3	Vinay	Jariwala	35100	3
4	Jagruti	Viras	9500	2
5	Shweta	Rana	12000	3
6	Sonal	Menpara	13000	1
7	Yamini	Patel	10000	2
8	Khyati	Shah	50000	3
9	Shwets	Jariwala	19400	2

Data filtering:

SQLite DISTINCT Clause

The SQLite DISTINCT clause is used with SELECT statement to eliminate all the duplicate records and fetching only unique records.

It is used when you have multiple duplicate records in the table.

Syntax:

```
SELECT DISTINCT column1, column2,..... column FROM table_name WHERE [condition];
```

Example:

```
SELECT DISTINCT first_name FROM EMP_MASTER;
```

SQLite WHERE Clause

- SQLite WHERE clause is generally used with SELECT, UPDATE and DELETE statement to specify a condition while you fetch the data from one table or multiple tables.
- If the condition is satisfied or true, it returns specific value from table. You would use WHERE clause to filter the records and fetching only necessary records.
- WHERE clause is also used to filter the records and fetch only specific data.
- WHERE clause with several comparison and logical operators. like > , < , = , LIKE , NOT

Syntax:

```
SELECT column1, column2, columnN FROM table_name WHERE [condition]
```

Example:

```
SELECT * FROM emp_master WHERE salary>=9500 AND DEPT_ID>=2;
```

SQLite BETWEEN Clauses:

The SQLite BETWEEN is used to select those values that are within a range of two values.

Example:

```
SELECT * FROM STUDENT WHERE ID BETWEEN 5 AND 8;
```

Que) Explain SQLite LIKE Clause**Ans:**

- SQLite LIKE operator is used to match text values against a pattern using wildcards. If the search expression can be matched to the pattern expression, the LIKE operator will return true, which is 1. There are two wildcards used in conjunction with the LIKE operator –
- The percent sign (%) and The underscore (_)
- The percent sign represents zero, one, or multiple numbers or characters. The underscore represents a single number or character. These symbols can be used in combinations.

Syntax:

```
SELECT FROM table_name WHERE column LIKE 'XXXX%';
or
SELECT FROM table_name WHERE column LIKE '%XXXX%';
or
SELECT FROM table_name WHERE column LIKE 'XXXX_';
or
SELECT FROM table_name WHERE column LIKE '_XXXX';
or
SELECT FROM table_name WHERE column LIKE '_XXXX_';
```

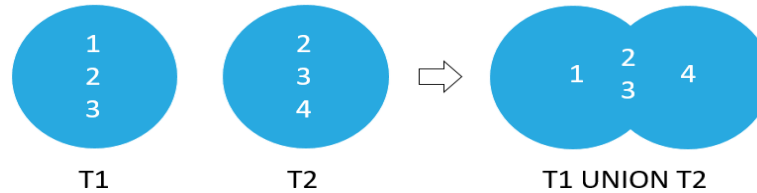
Example:

Statement	Description
WHERE SALARY LIKE '200%'	Finds any values that start with 200
WHERE SALARY LIKE '%200%'	Finds any values that have 200 in any position
WHERE SALARY LIKE '_00%'	Finds any values that have 00 in the second and third positions
WHERE SALARY LIKE '2_%_%'	Finds any values that start with 2 and are at least 3 characters in length
WHERE SALARY LIKE '%2'	Finds any values that end with 2
WHERE SALARY LIKE '_2%3'	Finds any values that has a 2 in the second position and ends with a 3
WHERE SALARY LIKE '2__3'	Finds any values in a five-digit number that starts with 2 and ends with 3

Que) Explain SQLite Union Operator

Ans. SQLite UNION clause is used to combine the results of two or more SELECT statements without returning any duplicate rows.

To use UNION, each SELECT must have the same number of columns selected, the same number of column expressions, the same data type, and have them in the same order, but they do not have to be of the same length.

**Syntax:**

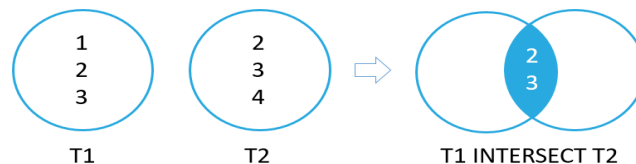
```
SELECT column1, column2.... FROM table1 WHERE condition;
UNION
SELECT column1, column2.... FROM table2 WHERE condition;
```

Example:

```
SELECT first_name, last_name, dept_id FROM emp_master WHERE dept_id = 1
UNION
SELECT first_name, last_name, dept_id FROM emp_master WHERE last_name = 'Patel';
```

Que) Explain SQLite- INTERSECT

Ans: SQLite INTERSECT operator compares the result sets of two queries and returns distinct rows that are output by both queries.

**Syntax:**

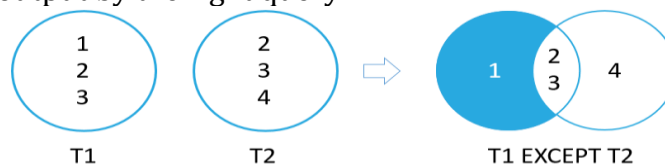
```
SELECT column1, column2.... FROM table1 WHERE condition;
INTERSECT
SELECT column1, column2.... FROM table2 WHERE condition;
```

Example:

```
SELECT last_name FROM emp_master WHERE dept_id = 2
INTERSECT
SELECT last_name FROM emp_master WHERE dept_id = 3;
```

Que) Explain SQLite – EXCEPT

Ans: SQLite EXCEPT operator compares the result sets of two queries and returns distinct rows from the left query that are not output by the right query.

**Syntax:**

```
SELECT column1, column2.... FROM table1
EXCEPT
SELECT column1, column2.... FROM table2;
```

Example:

```
SELECT first_name, last_name FROM emp_master WHERE dept_id = 2
EXCEPT
SELECT first_name, last_name FROM emp_master WHERE first_name = 'Shweta';
```

Que) Explain SQLite- LIMIT Clause

Ans: The SQLite LIMIT clause is used to limit the data amount fetched by SELECT command from a table.

Syntax:

```
SELECT column1, column2,.....columnN FROM table_name LIMIT[no of rows];
```

Example:

```
SELECT * FROM STUDENT LIMIT 3;
```

Que) Explain SQLite- GROUP BY Clause.

Ans: SQLite GROUP BY clause is used in collaboration with the SELECT statement to arrange identical data into groups.

GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Syntax:

```
SELECT column-list FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2....columnN
ORDER BY column1, column2....columnN
```

Example:

```
SELECT dept_id, COUNT(emp_id) AS total_employees FROM emp_master GROUP BY dept_id;
```

Que) Explain SQLite Null Values

Ans. Generally, all relational databases will support a special value called NULL and it is used to represent missing information. If table has Null value, then it will display as blank. The Null value represents the absence of a value or empty or no value. By using the NULL keyword, we can represent NULL or empty string values.

Que) Explain SQLite- CASE

Ans. The SQLite CASE expression evaluates a list of conditions and returns an expression based on the result of the evaluation.

- The CASE expression is similar to the IF-THEN-ELSE statement in other programming languages.
- you can use the CASE expression in clauses such as WHERE, ORDER BY, HAVING, SELECT and statements such as SELECT, UPDATE, and DELETE.

Syntax:

```
CASE case_expression
WHEN when_expression_1
    THEN result_1
WHEN when_expression_2
    THEN result_2
[ ELSE result_else ]
END;
```

Example

```

SELECT first_name, dept_id,
CASE dept_id
  WHEN 1 THEN 'HR'
  WHEN 2 THEN 'Sales'
  WHEN 3 THEN 'IT'
ELSE 'Unassigned'
END AS department_name
FROM emp_master;

```

Que) Explain SQLite- Joins

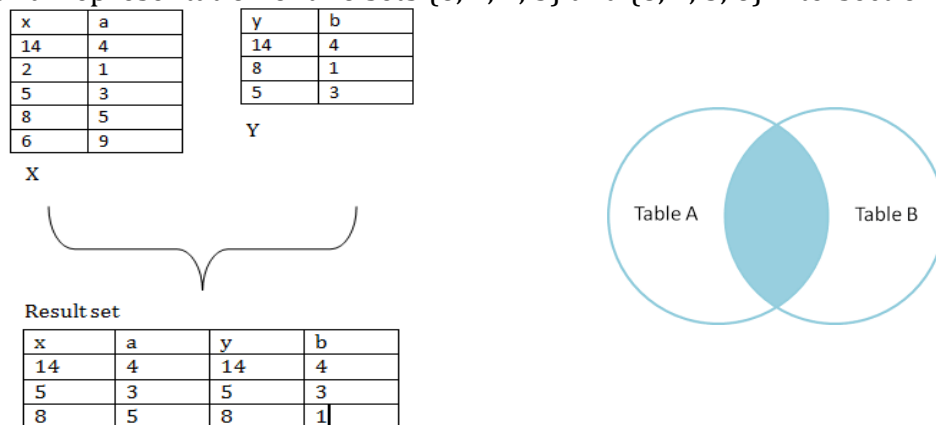
Ans: SQLite Joins clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

- SQL defines three major types of joins –
 1. The INNER JOIN
 2. The OUTER JOIN
 3. The CROSS JOIN

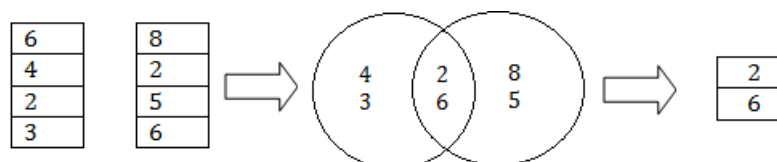
1. INNER JOIN

The SQLite inner join is the most common type of join. An INNER JOIN only returns rows where there **is a perfect match in both tables**. It is used to combine all rows from multiple tables where the join condition is satisfied.

Following is the pictorial representation of two sets {6, 4, 2, 3} and {8, 2, 5, 6} intersection operation.



Following is the simple pictorial representation of using SQLite Inner Join with two tables.



Syntax: SELECT columns FROM table1 [INNER] JOIN table2 ON table1.column = table2.column;

Example: Teacher Table:

	Id	name	salary
1	1	Rosey	35000
2	2	Jack	40000
3	3	Pink	20000
4	4	Mark	25000
5	5	Don	40000

Departments Table:

	Id	dept	emp_id
1	1	Biology	100
2	2	Maths	101
3	3	English	102
4	4	Social	103
5	5	GK	104

Example: SELECT t.name, d.dept FROM Teachers as t INNER JOIN department as d on t.Id = d.Id;

Output:

	name	dept
1	Rosey	Biology
2	Jack	Maths
3	Pink	English
4	Mark	Social
5	Don	GK

To create and insert some data in **dept_master** table execute following query.

```
CREATE TABLE dept_master (dept_id INTEGER PRIMARY KEY , dept_name TEXT NOT NULL);
```

```
INSERT INTO dept_master (dept_id, dept_name) VALUES (1, 'HR'), (2, 'IT'), (3, 'Finance'), (4, 'Marketing');
```

```
SELECT e.emp_id, e.first_name, e.last_name, e.salary, d.dept_name FROM emp_master e INNER JOIN dept_master d ON e.dept_id = d.dept_id;
```

```
SELECT d.dept_id, dept_name, emp_id, first_name FROM dept_master d JOIN emp_master e ON d.dept_id = e.dept_id;
```

2. OUTER JOIN:

- In SQLite, by using **Inner Join** we can get only matching rows from multiple tables based on the conditions defined in select statements. If we use SQLite **Outer Join** it will select matching rows from multiple tables same as Inner Join and some other rows outside of the relationship.
- **Three** types of Outer Joins in SQL standard those are LEFT, RIGHT and FULL Outer Joins but **SQLite supports only LEFT OUTER JOIN.**

Syntax: SELECT columns FROM table1 LEFT [OUTER] JOIN table2 ON table1.column = table2.column;

Example:

```
SELECT d.dept_id, dept_name, emp_id, first_name FROM dept_master d LEFT OUTER JOIN emp_master e ON d.dept_id = e.dept_id;
```

```
SELECT e.emp_id, e.first_name, e.last_name, e.salary, d.dept_name FROM emp_master e LEFT JOIN dept_master d ON e.dept_id = d.dept_id;
```

Example of LEFT OUTER JOIN:

```
SELECT t.name, t.salary, d.dept FROM Teachers as t LEFT OUTER JOIN department as d on t.Id = d.dept_id;
```

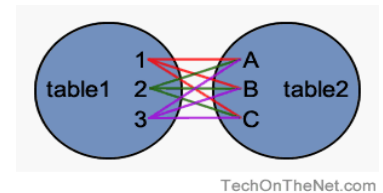
Output:

	name	salary	dept
1	Rosey	35000	NULL
2	Jack	40000	NULL
3	Pink	20000	NULL
4	Mark	25000	NULL
5	Don	40000	NULL

3. CROSS JOIN

Ans: In SQLite, CROSS JOIN is used to get the Cartesian product of rows by matching each row of the first table with every row of the second table.

Syntax: SELECT columns FROM table1 CROSS JOIN table2;



Example:

```
SELECT dept_name, emp_id, first_name FROM dept_master CROSS JOIN emp_master;
```

```
SELECT e.emp_id, e.first_name, e.last_name, d.dept_name FROM emp_master e CROSS JOIN dept_master d;
```

Example of CROSS JOIN: SELECT * FROM Teachers CROSS JOIN department;

Output:

	Id	name	salary	Id	dept	emp_id
1	1	Rosey	35000	1	Biology	100
2	1	Rosey	35000	2	Maths	101
3	1	Rosey	35000	3	English	102
4	1	Rosey	35000	4	Social	103
5	1	Rosey	35000	5	GK	104
6	2	Jack	40000	1	Biology	100
7	2	Jack	40000	2	Maths	101
8	2	Jack	40000	3	English	102
9	2	Jack	40000	4	Social	103
10	2	Jack	40000	5	GK	104
11	3	Pink	20000	1	Biology	100
12	3	Pink	20000	2	Maths	101
13	3	Pink	20000	3	English	102
14	3	Pink	20000	4	Social	103
15	3	Pink	20000	5	GK	104
16	4	Mark	25000	1	Biology	100
17	4	Mark	25000	2	Maths	101
18	4	Mark	25000	3	English	102
19	4	Mark	25000	4	Social	103
20	4	Mark	25000	5	GK	104
21	5	Don	40000	1	Biology	100
22	5	Don	40000	2	Maths	101
23	5	Don	40000	3	English	102
24	5	Don	40000	4	Social	103
25	5	Don	40000	5	GK	104

Que) Explain SQLite- IN

Ans: The SQLite IN operator determines whether a value matches any value in a list or a subquery.

Syntax:

```
expression IN (val1, val2,... val_n);
OR
expression IN (subquery);
```

- The expression can be any valid expression or a column of a table.
- A list of values is a fixed value list or a result set of a single column returned by a subquery.

Example:

```
SELECT * from emp_master WHERE dept_id
IN (SELECT dept_id from dept_master
WHERE dept_name= 'HR');
```

Que) SQLite-HAVING CLAUSE

Ans: HAVING clause enables you to specify conditions that filter which group results appear in the final results.

Syntax: SELECT column1, column2 FROM table1, table2 WHERE [conditions]
GROUP BY column1, column2
HAVING [conditions]
ORDER BY column1, column2

Example: SELECT * FROM COMPANY GROUP BY name HAVING count(name) < 2;

Que) Explain SQLite-TRIGGER

Ans: In SQLite trigger is a database object and it will raise automatically whenever an insert, update and delete operations performed on a particular table or view in the database.

Benefits of Trigger in SQLite

- It will help us to prevent invalid transactions by imposing business rules
- Maintain data integrity
- We can enable event logging for the changes in table data.
- NEW and OLD reference (new is used in insert update and delete, old is used in update).

Syntax:

```
CREATE TRIGGER [IF NOT EXISTS] trigger_name [BEFORE|AFTER|INSTEAD OF]
[INSERT|UPDATE|DELETE]
ON table_name
[FOR EACH ROW | FOR EACH STATEMENT] [WHEN condition]
BEGIN
    Trigger_action_body
END;
```

Example: Create a trigger

-- Main table

```
CREATE TABLE employees ( id INTEGER PRIMARY KEY, name TEXT NOT NULL, salary REAL NOT NULL);
```

-- Table to store salary changes

```
CREATE TABLE salary_history (employee_id INTEGER, old_salary REAL, new_salary REAL, changed_at TEXT DEFAULT CURRENT_TIMESTAMP );
```

Create the Trigger:

Next, create an AFTER UPDATE trigger. It will look for changes specifically on the salary column and insert the old and new values into salary_history.

```
CREATE TRIGGER emp_trig AFTER UPDATE OF salary ON employees
BEGIN
INSERT INTO salary_history (employee_id, old_salary, new_salary)
VALUES (OLD.id, OLD.salary, NEW.salary);
END;
```

Key Keywords:

- AFTER UPDATE OF salary: Fire the trigger *only after* the salary column is modified.
- OLD: A special alias representing the row data *before* the update.
- NEW: A special alias representing the row data *after* the update.

Testing the Trigger**Step A: Insert a record**

```
INSERT INTO employees (name, salary) VALUES ('Pooja Soni', 50000.00);
```

Step B: Update the salary

```
UPDATE employees SET salary = 55000.00 WHERE id = 1;
```

Step C: Verify the results

```
SELECT * FROM salary_history;
```

How to Delete a Trigger

```
DROP TRIGGER IF EXISTS log_salary_update;
```

Create trigger after Insert:

```
CREATE TRIGGER log_new_employee_salary  
AFTER INSERT ON employees  
FOR EACH ROW  
BEGIN  
    INSERT INTO salary_history (employee_id, old_salary, new_salary)  
    VALUES (NEW.id, 0.0, NEW.salary);  
END;
```

```
SELECT * FROM employees;
```

Create trigger before Insert:

```
INSERT INTO employees (name, salary) VALUES ('Aarana Patel', -50000.00);
```

```
CREATE TRIGGER validate_employee_salary_before_insert  
BEFORE INSERT ON employees  
FOR EACH ROW  
BEGIN  
    SELECT  
        CASE  
            WHEN NEW.salary < 0 THEN  
                RAISE(ABORT, 'Error: Salary cannot be negative.')
```

Create trigger before Update:

```
CREATE TRIGGER enforce_no_pay_cuts
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
    SELECT
        CASE
            WHEN NEW.salary < OLD.salary THEN
                RAISE(ABORT, 'Error: Salary reductions are not allowed.')
        END;
END;
```

Create trigger after Update:

```
CREATE TRIGGER log_salary_history_update
AFTER UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
    INSERT INTO salary_history (employee_id, old_salary, new_salary)
    VALUES (OLD.id, OLD.salary, NEW.salary);
END; END;
```

Create trigger before delete:

```
CREATE TRIGGER protect_high_earners
BEFORE DELETE ON employees
FOR EACH ROW
BEGIN
    SELECT
        CASE
            WHEN OLD.salary > 100000.00 THEN
                RAISE(ABORT, 'Error: Cannot delete high-earning employees via standard commands.')
        END;
END;
```

create trigger after delete

```
CREATE TRIGGER log_employee_deletion
AFTER DELETE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO salary_history (employee_id, old_salary, new_salary)
    VALUES (OLD.id, OLD.salary, 0.0);
END;
```

Step 1: Set Up the Base Tables

Before creating triggers, you need the underlying tables: one for the current employee records and one to log the salary history.

1. Create the employees table

```
CREATE TABLE employees (  
    id INTEGER PRIMARY KEY AUTOINCREMENT,  
    name TEXT NOT NULL,  
    salary REAL NOT NULL  
);
```

2. Create the salary history log table

```
CREATE TABLE salary_history (  
    history_id INTEGER PRIMARY KEY AUTOINCREMENT,  
    employee_id INTEGER,  
    old_salary REAL,  
    new_salary REAL,  
    change_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
);
```

Step 2: Implement "BEFORE" Triggers (Validation & Protection)

These triggers act as guards. They check data *before* it gets written to the database and will abort the operation if rules are broken.

A. Prevent Negative Salaries (Before Insert)

```
CREATE TRIGGER validate_employee_salary_before_insert  
BEFORE INSERT ON employees  
FOR EACH ROW  
BEGIN  
    SELECT  
        CASE  
            WHEN NEW.salary < 0 THEN  
                RAISE(ABORT, 'Error: Salary cannot be negative.')  
        END;  
END;
```

B. Prevent Pay Cuts (Before Update)

```
CREATE TRIGGER enforce_no_pay_cuts  
BEFORE UPDATE OF salary ON employees  
FOR EACH ROW  
BEGIN  
    SELECT  
        CASE  
            WHEN NEW.salary < OLD.salary THEN  
                RAISE(ABORT, 'Error: Salary reductions are not allowed.')  
        END; END;
```

C. Protect High Earners from Deletion (Before Delete)

```
CREATE TRIGGER protect_high_earners
BEFORE DELETE ON employees
FOR EACH ROW
BEGIN
    SELECT
        CASE
            WHEN OLD.salary > 100000.00 THEN
                RAISE(ABORT, 'Error: Cannot delete high-earning employees via standard commands.')
            END;
END;
```

Step 3: Implement "AFTER" Triggers (Auditing & Logging)

These triggers automatically log history *after* a successful database operation has occurred.

A. Log New Hires (After Insert)

```
CREATE TRIGGER log_new_employee_salary
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
    INSERT INTO salary_history (employee_id, old_salary, new_salary)
    VALUES (NEW.id, 0.0, NEW.salary);
END;
```

B. Log Salary Changes (After Update)

```
CREATE TRIGGER log_salary_history_update
AFTER UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
    INSERT INTO salary_history (employee_id, old_salary, new_salary)
    VALUES (OLD.id, OLD.salary, NEW.salary);
END;
```

C. Log Employee Departures (After Delete)

```
CREATE TRIGGER log_employee_deletion
AFTER DELETE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO salary_history (employee_id, old_salary, new_salary)
    VALUES (OLD.id, OLD.salary, 0.0);
END;
```

Step 4: Test the System

Run these queries sequentially to see both the validation guardrails and the audit logging in action.

Scenario A: Testing Insertion & Validation**1. Try inserting a negative salary (Should Fail):**

```
INSERT INTO employees (name, salary) VALUES ('Aarana Patel', -50000.00);
```

Result: Aborted with 'Error: Salary cannot be negative.'

2. Insert a valid employee record (Should Pass):

```
INSERT INTO employees (name, salary) VALUES ('Pooja patel', 56000.00);
```

3. Verify salary_history:

```
SELECT * FROM salary_history;
```

Result: You will see a row tracking Pooja's initial entry (Old: 0.0, New: 50000.0).

Scenario B: Testing Updates & Pay Cut Prevention**1. Try giving a pay cut (Should Fail):**

```
UPDATE employees SET salary = 45000.00 WHERE name = 'Pooja Soni';
```

Result: Aborted with 'Error: Salary reductions are not allowed.'

2. Give a legitimate raise (Should Pass):

```
UPDATE employees SET salary = 55000.00 WHERE name = 'Pooja Soni';
```

3. Verify the log history again:

```
SELECT * FROM salary_history;
```

Result: A second entry appears showing the jump from 50000.0 to 55000.0.

Scenario C: Testing Deletions & High-Earner Protection**1. Insert a high earner:**

```
INSERT INTO employees (name, salary) VALUES ('Rich Boss', 120000.00);
```

2. Try deleting the high earner (Should Fail):

```
DELETE FROM employees WHERE name = 'Rich Boss';
```

Result: Aborted with 'Error: Cannot delete high-earning employees...'

3. Delete a standard earner (Pooja) (Should Pass):

```
DELETE FROM employees WHERE name = 'Pooja Soni';
```

4. Verify final logs:

```
SELECT * FROM salary_history;
```

Result: The final log entry shows Pooja's deletion (Old: 55000.0, New: 0.0).

Step 5: How to Clean Up / Delete Triggers

If you ever need to alter the logic or remove a trigger entirely, use the DROP TRIGGER command:

```
DROP TRIGGER IF EXISTS validate_employee_salary_before_insert;
DROP TRIGGER IF EXISTS enforce_no_pay_cuts;
DROP TRIGGER IF EXISTS protect_high_earners;
DROP TRIGGER IF EXISTS log_new_employee_salary;
DROP TRIGGER IF EXISTS log_salary_history_update;
DROP TRIGGER IF EXISTS log_employee_deletion;
```

Checking an AFTER INSERT Trigger

This trigger is supposed to automatically create a log entry in salary_history when a new employee is added.

- **Step A: Run the operation**

```
INSERT INTO employees (name, salary) VALUES ('John Doe', 60000.00);
```

- **Step B: Check the result** Query the log table specifically looking for this new employee's ID (assuming it generated id = 2):

```
SELECT * FROM salary_history WHERE employee_id = 2;
```

- **What you should see:** A single row showing old_salary as 0.0 and new_salary as 60000.00.

2. Checking a BEFORE INSERT (Validation) Trigger

This trigger is designed to block data from entering the database if it breaks a rule (like a negative salary).

- **Step A: Run the invalid operation**

```
INSERT INTO employees (name, salary) VALUES ('Invalid User', -1000.00);
```

- **Step B: Check the result**

```
SELECT * FROM employees WHERE name = 'Invalid User';
```

- **What you should see:** The database will throw an explicit error window stating Error: Salary cannot be negative. If you run the SELECT query, it will return **0 rows**, proving the trigger successfully blocked the insert.

3. Checking an AFTER UPDATE Trigger

This trigger fires only when a specific column (salary) is changed, logging the transition from old to new.

- **Step A: Run the operation**

```
UPDATE employees SET salary = 65000.00 WHERE name = 'John Doe';
```

- **Step B: Check the result**

```
SELECT * FROM salary_history WHERE employee_id = 2;
```

4. Checking a BEFORE DELETE Trigger

This trigger blocks a high-earner from being accidentally removed.

- **Step A: Setup the condition**

```
INSERT INTO employees (name, salary) VALUES ('CEO Mike', 150000.00);
```

- **Step B: Attempt the restricted operation**

```
DELETE FROM employees WHERE name = 'CEO Mike';
```

- **Step C: Check the result**

```
SELECT * FROM employees WHERE name = 'CEO Mike';
```

Pro-Tip for Testing

See current state of employees

```
SELECT * FROM employees;
```

See historical tracking logs

```
SELECT * FROM salary_history;
```