Hewlett Packard Enterprise

HPE CTY

Network Flow Database

A Report On

SQLite

Documented By:
Monika K

Introduction:

SQLite is a lightweight, self-contained, server-less database engine that is widely used for embedded database applications. Unlike traditional relational database management systems (RDBMS) that operate with a client-server model, SQLite operates in-process with the application, which means the database engine runs within the same program that uses it. This architecture makes SQLite an ideal choice for applications that require a simple, fast, and reliable way to manage data without the overhead of a separate database server.

First released in 2000, SQLite has grown to become one of the most deployed database engines in the world, thanks to its simplicity, efficiency, and robustness. It is embedded in a multitude of applications, including web browsers, mobile phones, operating systems, and various other software systems. SQLite's source code is in the public domain, allowing for unrestricted use and distribution, which has further contributed to its widespread adoption.

Description of SQLite and Use-cases:

One of SQLite's standout features is its simplicity. The entire database is stored in a single file on disk, which simplifies both deployment and backup processes. Despite its compact size, SQLite supports a rich set of features, including most of the SQL92 standard, transactions, and multi-user access, making it a powerful tool for managing data in a variety of contexts.

In addition to its ease of use, SQLite is highly efficient and performs well even on constrained hardware. It supports standard SQL syntax and transactions, ensuring data integrity and reliability. Its small footprint (typically less than 1 MB) and low resource consumption make it ideal for embedded systems and mobile applications.

Key Features of SQLite:

- Zero Configuration: No setup or administration required.
- Serverless: SQLite runs directly in the application, eliminating the need for a separate database server.
- Self-Contained: A single library, with no dependencies on external libraries.
- Cross-Platform: Runs on numerous platforms, including Windows, Mac OS, Linux, iOS, and Android.
- Transactional: Implements ACID (Atomicity, Consistency, Isolation, Durability) properties for reliable transactions.
- Compact: The library size is minimal, making it suitable for embedded applications.
- Public Domain: Free for use, distribution, and modification.

Use-cases:

1. Web Browsers

SQLite is embedded in many web browsers, including Google Chrome, Mozilla Firefox, and Apple Safari, where it is used to manage local storage of data. In these browsers, SQLite databases are used to store user preferences, bookmarks, history, and offline data, enabling fast and efficient access to data without the need for constant network requests.

2. Mobile Applications

Many mobile applications use SQLite to manage local data storage. For instance, messaging apps, email clients, and social media apps utilize SQLite to store messages, user profiles, and other data locally. This allows the application to function smoothly even without a constant internet connection, improving user experience and app performance.

3. Network Configuration and Management Tools

Network configuration tools often use SQLite to store configuration data, logs, and other critical information. By using SQLite, these tools can quickly read and write configurations, ensuring that changes are applied swiftly and accurately. An example is the Network Manager tool on Linux systems, Network Manager doesn't directly use SQLite, but it provides essential tools for managing network connections on Linux systems

4. IoT Devices

In the realm of Internet of Things (IoT), SQLite is widely used to manage data on devices that are intermittently connected to the network. For example, sensors and smart home devices often store collected data in SQLite databases locally. When network connectivity is available, the data can be synchronized with a central server or cloud service.

5. Server-Side Database:

While not as common, SQLite can also be used for smaller web applications or internal tools where the overhead of a full database server is not justified.

6. Testing and Prototyping:

Developers use SQLite to quickly set up a test database without the overhead of installing and configuring a server-based database system. This is useful for development, testing, and proof-of-concept projects.

Design of network flow database:

Structure:

A Network Flow database is designed to efficiently capture, store, and analyze packet information passing through the switches in the network. This allows users to efficiently store, retrieve, and modify information about different packets sent in a network.

The Network flow database contains the following attributes:

1) **"source_ip"**: This field is used to store the source IP address of the packet.
2) **"destination_ip"**: This field is used to store the destination IP address of the packet.
3) **"source_port"**: This field stores the source port number of the application.
4) **"destination_port"**: This field stores the destination port number.
5) **"version"**: (IPv4 or IPv6) This stores the protocol used in the network layer.

To avoid redundancy, all five attributes should be unique in the database. Thus, any packet containing all five-attribute information is stored in the network flow database and used as per the client's requirements.

Reasons that it fits for implementing the network flow database:

SQLite is a suitable choice for implementing a network flow database due to several key reasons:

- **Lightweight and Embedded:** SQLite is a lightweight, self-contained database engine that operates in-process with the application. This means that it doesn't require a separate server process to run, making it easy to embed within network monitoring tools or other applications without adding significant overhead.
- **Low Resource Consumption:** SQLite is designed to have a small memory footprint and low resource consumption, making it well-suited for environments with limited resources, such as embedded systems or network monitoring devices. This allows it to efficiently handle large volumes of network flow data without straining the system.
- **Efficient Storage and Retrieval**: SQLite uses a compact file-based storage format, which is optimized for efficient storage and retrieval of data. This is beneficial for storing network flow data, which can accumulate rapidly and require fast access for analysis and reporting purposes.
- **Transactional Support:** SQLite provides support for transactions, ensuring data integrity and consistency even in the event of system failures or interruptions. This is crucial for network flow databases, where accurate and reliable data capture is essential for monitoring and analyzing network traffic.
- **SQL Compatibility:** SQLite supports a subset of SQL commands, allowing developers to perform complex queries, filtering, and aggregation operations on network flow data. This enables advanced analysis and reporting capabilities, facilitating the identification of trends, anomalies, and security threats within the network.
- **Cross-Platform Compatibility:** SQLite databases can be easily shared and accessed across different platforms and operating systems, including Linux, Windows, mac-OS, and various embedded systems. This flexibility allows network flow data to be collected and analyzed on diverse infrastructure environments.

- **Community Support and Documentation:** SQLite has a large and active community of users and developers, providing extensive documentation, tutorials, and resources for implementing and optimizing SQLite databases. This support network can be invaluable for developers working on network flow database projects.

Advantages of SQLite:

No Administration Required: Unlike traditional database systems, SQLite requires no administration. You don't need a separate database server or complex configurations1.

Low-to-Medium Traffic HTTP Requests: SQLite is particularly well-suited for handling low-to-medium traffic HTTP requests. It efficiently manages concurrent connections without the need for extensive setup1.

Lightweight and Portable: SQLite is a lightweight library that can be integrated seamlessly into applications. It's ideal for mobile or IoT applications with limited resources. Plus, it's portable across different operating systems and architectures2.

Reliable and Self-Contained: SQLite ensures data integrity through transactions. It's a self-contained database engine, eliminating the need for external dependencies or maintenance by a Database Administrator3.

Fast and Efficient: Reading and writing operations are fast, making it suitable for scenarios where performance matters. It only loads necessary data, minimizing memory usage1.

Full-Text Indexing : SQLite supports full-text indexing for efficient text searches.

Storing Configuration Data: SQLite is an excellent choice for storing configuration data for a program. It provides a fast and powerful way to store this data, making it easy to access and manipulate as needed.

Disadvantages of SQLite:

Limited Database Size: SQLite databases are limited to 2GB in size. If the application requires larger storage, consider other databases.

Concurrency Limitations: While SQLite handles multiple readers efficiently, write concurrency can be a bottleneck under heavy loads.

Not Suitable for High-Traffic Servers: It's not designed for high-traffic web servers or large-scale enterprise applications.

No Network Access: Since SQLite is a file-based database, it doesn't support network access. It's not suitable for client-server architectures.

Less Optimized for Write-Heavy Workloads: If your application primarily involves frequent writes, consider other databases optimized for write-heavy scenarios.

SQLite features:

Relational/ Non-Relational?	In memory /Disk based (RAM v/s SSD)	Publisher-Subscriber notification support for row/column updates (Y/N)	Partial key search support (Y/N)	Open source License used	Client Library language support	Multi-threaded publisher/subscriber support
Relational	Both	No	Yes	SQLite is in the public domain (Public Domain License)	Python, Java, JavaScript, C,C++, Ruby, Go, php, swift, perl	Has Multi-thread support

1. Relational/Non-Relational:
   Relational: SQLite is a relational database management system (RDBMS). It follows the relational model, supporting tables with rows and columns, primary keys, and SQL queries.
   Non-Relational: While SQLite primarily fits the relational model, it can also be used for non-relational purposes due to its flexibility.

2. In Memory/Disk Based (RAM vs. SSD):
   By default, SQLite databases are disk-based, meaning they persist on storage (SSD or HDD).
   However, you can create an in-memory database by specifying a special filename (:memory:) during connection. In-memory databases reside entirely in RAM and are volatile (lost when the process ends).

3. Publisher-Subscriber Notification Support for Row/Column Updates:
   No built-in publisher-subscriber support: SQLite doesn't natively provide publisher-subscriber notification mechanisms for row/column updates. You'd need to implement custom logic or use external tools/libraries for such notifications

4. Partial Key Search Support:
   Yes: SQLite supports partial key search using the LIKE operator with wildcard characters (% and _). It allows pattern matching within text columns.

5. Open Source License Used:
   SQLite uses a public domain-like license. The core SQLite software is in the public domain, but licenses are available for specific needs.

6. Client Library Language Support:
   SQLite has client libraries for various languages, including:
   - Python (sqlite3)
   - JavaScript (sql.js)
   - Java (sqlite-jdbc)
   - Ruby (sqlite3-ruby)

- Go (go-sqlite3)
- PHP (PDO SQLite)
- Swift (SQLite.swift)
- Perl (DBD::SQLite)

7. Multi-Threaded Publisher/Subscriber Support: SQLite supports multi-threading but doesn't natively provide publisher-subscriber features. Custom synchronization is required for multi-threaded access.

Additional SQLite features:

In SQLite, the PRAGMA statement is used to modify the operation of the SQLite library or to query the internal state of the library. Pragmas are a key feature of SQLite that provide a flexible mechanism to adjust various database settings and retrieve internal information. They can be used for a wide range of purposes, such as configuring database performance settings, checking the integrity of the database, and controlling aspects of the database's operation.

Here are the PRAGMA which can be used for optimization for our usecase:

By adjusting these PRAGMA settings, we can optimize SQLite performance for our specific use case:

1. cache_size: Larger cache reduces disk I/O.
2. temp_store: Using memory for temporary storage speeds up operations.
3. synchronous: NORMAL mode balances safety and performance.
4. journal_mode: WAL mode improves concurrency and write performance.
5. page_size: Larger page size improves performance for larger records.

Performance testing of SQLite using python modules

The dataset used in performance measurement, is prepared by a Python script which generates random unique IP address and port number pairs, simulating real network packet exchanges. It includes functions to generate random IPv4 and IPv6 addresses, random ports, and to randomly select the IP type. The script initializes parameters, tracks unique pairs with a dictionary, and uses a counter to ensure a large number of unique request-response pairs. Each pair and its reverse (representing the response packet) is written to a CSV file, ensuring unique and realistic network packet data. Python codes described below uses these csv files generated.

1. Normal operation:

Description: The code written is a Python script that manages a SQLite database to store and manipulate network flow data. It handles inserting, updating, and deleting records based on data from CSV files individually.

Here is a step-by-step breakdown of the code.

Importing Libraries:

- sqlite3: To interact with the SQLite database.
- pandas: To handle CSV file reading and manipulation.
- time: To measure execution time.
- matplotlib.pyplot: For potential future use in plotting data.

Function insert_data:

- Arguments: table_name, csv_file.
- Purpose: Drops an existing table (if any), creates a new table, and inserts data from the specified CSV file.

Steps:

- Connects to the SQLite database flow.db.
- Drops the table if it exists.
- Creates a new table with the specified schema (primary key includes source_ip, destination_ip, source_port, destination_port, and version).
- Reads the CSV file into a pandas DataFrame.
- Iterates through the DataFrame, inserting each row into the database.
- Tracks and prints the total time taken for insertion.

Function update_data:

- Arguments: table_name, csv_file.
- Purpose: Updates source_port to 100 for matching records in the specified table.

Steps:

- Connects to the SQLite database flow.db.

- Reads the CSV file into a pandas DataFrame.
- Iterates through the DataFrame, updating the matching records in the database.
- Tracks and prints the total time taken for updating.

Function delete_data:

- Arguments: table_name, csv_file.
- Purpose: Deletes matching records from the specified table based on the CSV file.

Steps:

- Connects to the SQLite database flow.db.
- Reads the CSV file into a pandas DataFrame.
- Iterates through the DataFrame, deleting the matching records in the database.
- Tracks and prints the total time taken for deletion.

Main Loop:

- Provides a user interface to choose between inserting, updating, and deleting data.
- Calls the appropriate function based on user input.
- Handles invalid inputs and allows the user to exit the loop by typing 'exit'.

Code link:- https://github.com/MonikaK2409/SQLite/blob/main/Codes/normal_operations.py

2. Batch operation:

Description:

The provided code is a Python script designed to manage a SQLite database that stores network flow data. It offers functionality to insert, update, and delete records in the database using batch processing for improved efficiency. The script uses the sqlite3 module for database interactions.

Importing Libraries:

- sqlite3: Provides a lightweight disk-based database, which doesn't require a separate server process.
- pandas: A powerful data manipulation and analysis library, used here for reading CSV files.
- time: Used to measure the time taken for database operations.
- matplotlib.pyplot: For potential future use in plotting data.

Function insert_data_batch:

- Arguments: table_name, csv_file, batch_size.
- Purpose: Drops an existing table if it exists, creates a new table, and inserts data from the specified CSV file in batches.

Steps:

- Connects to the SQLite database flow.db.
- Drops the table if it exists and creates a new table with specified columns and primary key constraints.
- Reads the CSV file in chunks specified by batch_size and inserts the data into the table.
- Tracks and prints the total time taken for insertion.

Function delete_data:

- Arguments: table_name, csv_file, batch_size.
- Purpose: Deletes records from the specified table in batches based on the data from the CSV file.

Steps:

- Connects to the SQLite database flow.db.
- Reads the entire CSV file into a DataFrame.
- Divides the data into batches and executes batch deletion operations.
- Tracks and prints the total time taken for deletion.
- Uses transactions to ensure that operations are atomic and can be rolled back in case of an error

.

Function update_data:

- Arguments: table_name, csv_file, batch_size.
- Purpose: Updates records in the specified table in batches based on the data from the CSV file.

Steps:

- Connects to the SQLite database flow.db.
- Reads the entire CSV file into a DataFrame.
- Divides the data into batches and executes batch update operations.
- Updates the source_port to 100 for matching records.
- Tracks and prints the total time taken for updating.
- Uses transactions to ensure that operations are atomic and can be rolled back in case of an error.

Main Loop:

- Provides a simple user interface to choose between inserting, updating, and deleting data.
- Prompts the user to enter the batch size for operations.
- Calls the appropriate function based on user input.
- Allows the user to exit the loop by typing 'exit'.
- Handles invalid inputs by prompting the user to select a valid option.

Code link:- https://github.com/MonikaK2409/SQLite/blob/main/Codes/batch_operations.py

Results

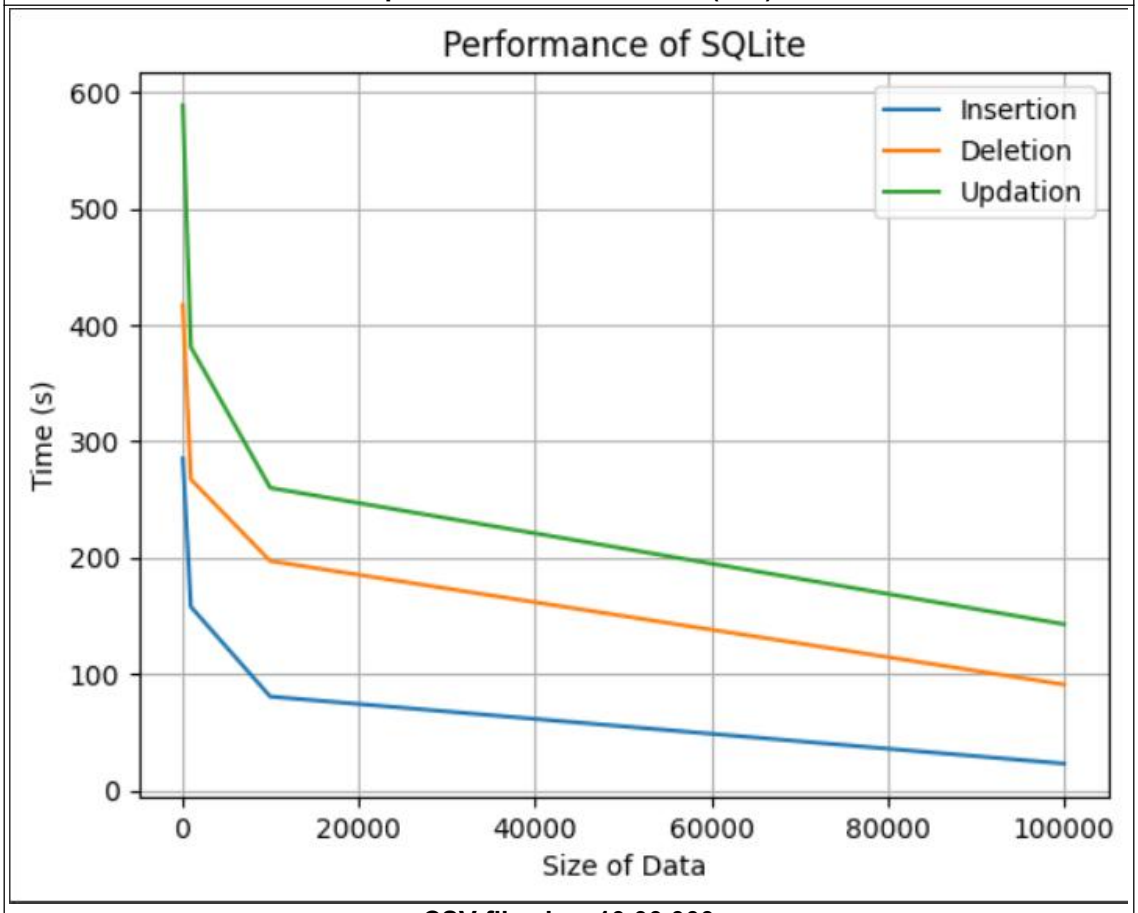
System Overview: we have used a System with 8GB RAM, and 1TB storage. All the data is recorded using the UBUNTU operating system (version 22.04).

Performance of batch operations:

When the python script for batch operations was executed the following output was obtained.

1. Data size of 10lakh and batchsizes of 100, 1000, 10000, 100000

Operation times for SQLite (in s)

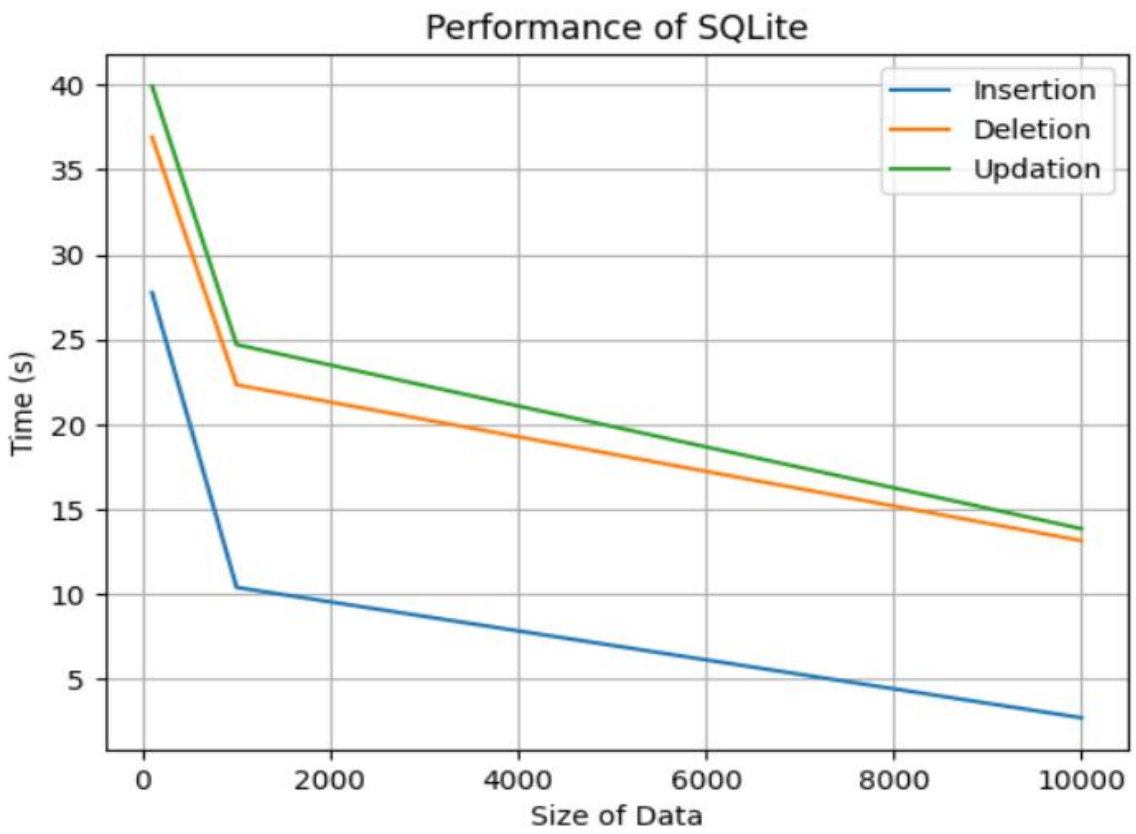


CSV file size=10,00,000

Batch_size	100	1000	10000	100000
Insertion_time	285.2	157.74	80.53	22.92
Deletion_time	417.12	267.26	197.03	90.87
Updation_time	588.71	380.57	259.95	142.75

2. Data size of 1lakh and batchsizes of 100, 1000, 10000

Operation times for SQLite (in s)

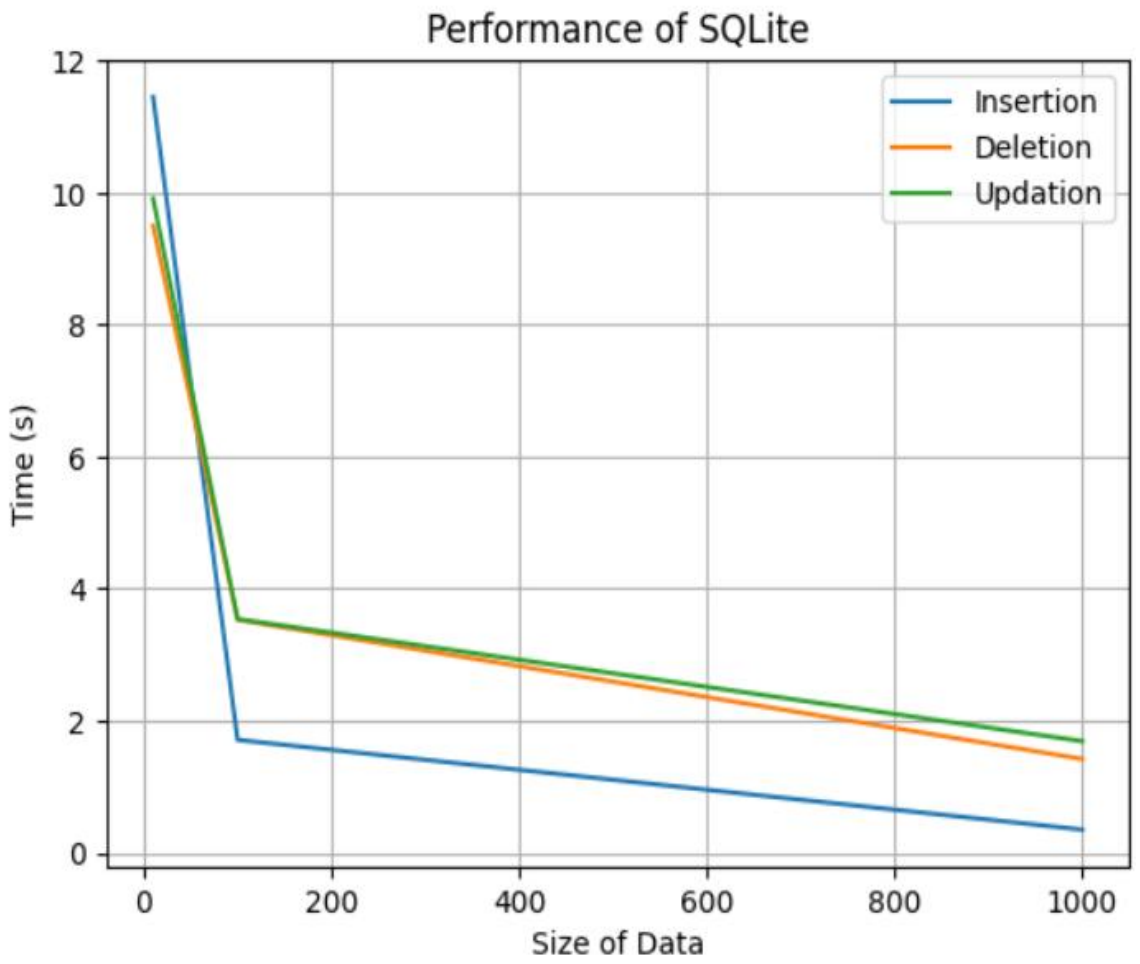


CSV file size=1,00,000

Batch_size	100	1000	10000
Insertion_time	27.75	10.38	2.7
Deletion_time	36.91	22.32	13.14
Updation_time	39.93	24.69	13.84

3. Data size of 10000 and batchsizes of 10, 100, 1000

Operation times for SQLite (in s)



CSV file size=10,000

Batch_size	**10**	**100**	**1000**
Insertion_time	11.45	1.71	0.35
Deletion_time	9.5	3.53	1.42
Updation_time	9.91	3.54	1.69

Introduction to YCSB

The Yahoo! Cloud Serving Benchmark (YCSB) is an open-source database bench-marking suite and a critical analytical component of cloud-based database management system (DBMS) evaluation. It allows users to comparatively measure how various modern SQL and NoSQL DBMS perform simple database operations on generated datasets.

We used the JDBC binding in Yahoo Cloud serving benchmark to execute the queries in SQLite.
The set-up is as described in the link :- https://github.com/MonikaK2409/SQLite/tree/main/Task-05

The structure of the default table in SQLite jdbc – binding is as follows:

```
CREATE TABLE usertable (
YCSB_KEY VARCHAR (255) PRIMARY KEY,
FIELD0 TEXT, FIELD1 TEXT,
FIELD2 TEXT, FIELD3 TEXT,
FIELD4 TEXT, FIELD5 TEXT,
FIELD6 TEXT, FIELD7 TEXT,
FIELD8 TEXT, FIELD9 TEXT
);
```

These are the types of workloads in YCSB

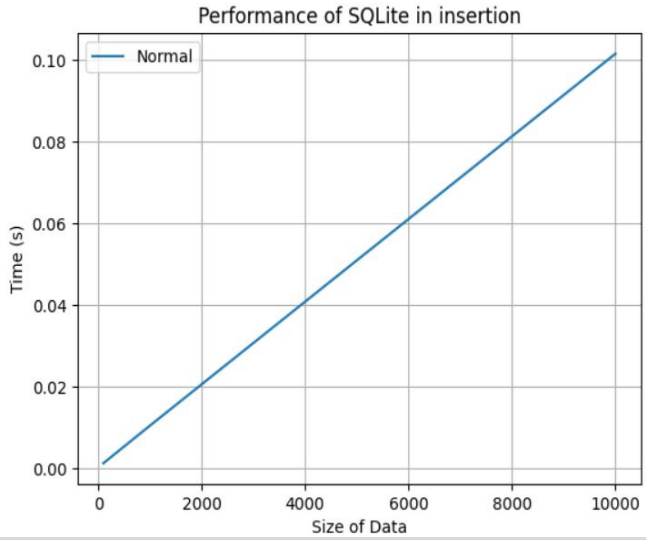
- Workload A: Update heavy workload: 50/50% Mix of Reads/Writes
- Workload B: Read mostly workload: 95/5% Mix of Reads/Writes
- Workload C: Read-only: 100% reads
- Workload D: Read the latest workload: More traffic on recent inserts
- Workload E: Short ranges: Short range-based queries
- Workload F: Read-modify-write: Read, modify, and update existing records

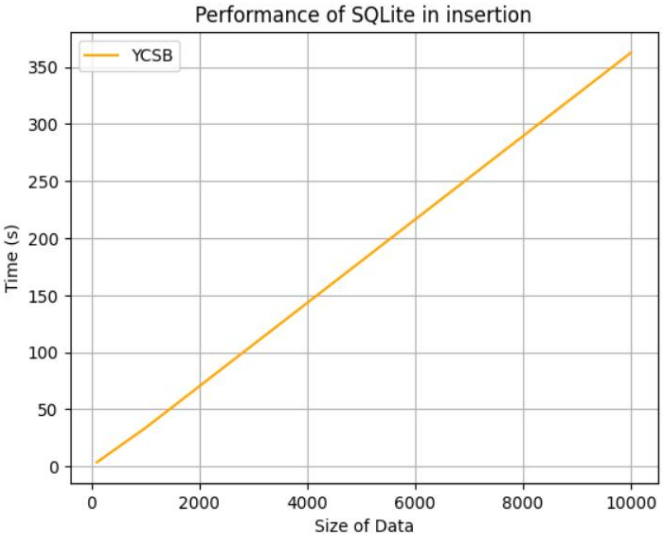
The results of all these workloads are given in the following link:
https://github.com/MonikaK2409/SQLite/tree/main/Task-07/YCSB_default_workloads_results

Performance Comparison of Normal operations and YCSB:

1. Insertion of tuples of size 100,1000,10000

Operations times for SQLite (in s)			
Data size	**100**	**1000**	**100000**
YCSB	0.0012	0.0104	0.1015
Normal_operation.py	3.45	33.58	362.55

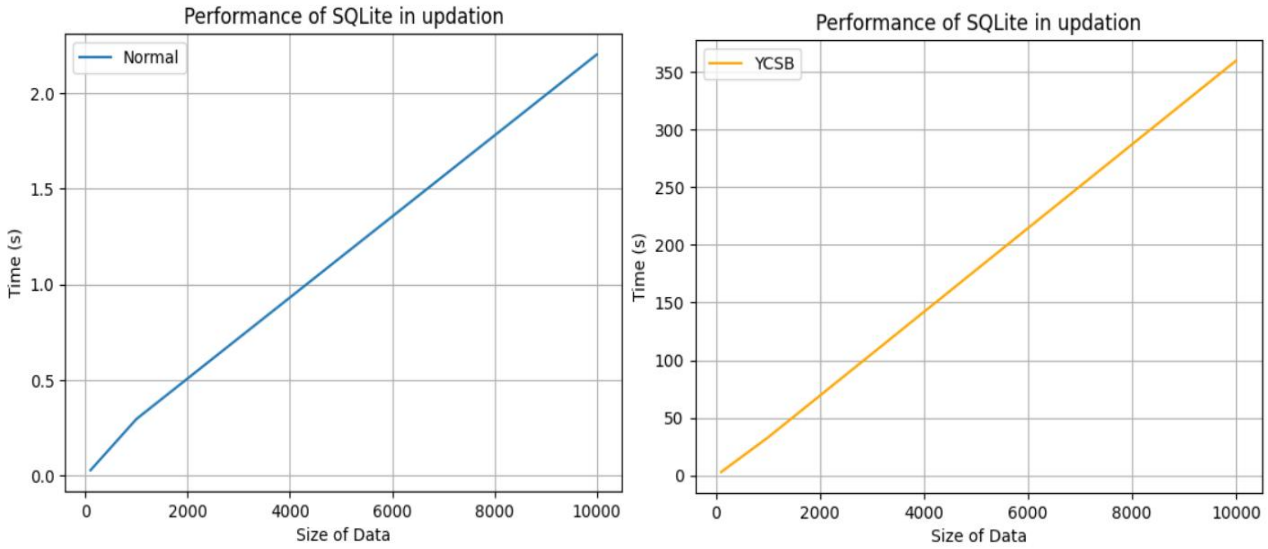




- Both YCSB and the Python script show a linear increase in the plotted values.
- The ratio between corresponding values from YCSB and the Python script remains nearly constant, for example, $0.0104/0.0012 \approx 9$ and $33.58/3.45 \approx 9$.
- Therefore, it can be concluded that both YCSB and the Python script are effectively conveying the same results.And our approach to do performance measure is verified.

2. Updation of tuples of size 100,1000,10000

Operations times for SQLite (in s)			
Data size	**100**	**1000**	**100000**
YCSB	0.02	0.2968	2.199
Normal_operation.py	2.9799	32.919	359.73



- Similarly, plotted values in updation from both YCSB and the Python script exhibit a linear increase.
- The ratio of corresponding values from YCSB to those from the Python script stays approximately constant.
- Hence, it can be concluded that YCSB and the Python script are consistently conveying similar results. Therefore our approach to do the performance measure is verified.

Additional Settings:

In SQLite, the PRAGMA statement is used to modify the operation of the SQLite library or to query the internal state of the library. Pragmas are a key feature of SQLite that provide a flexible mechanism to adjust various database settings and retrieve internal information. They can be used for a wide range of purposes, such as configuring database performance settings, checking the integrity of the database, and controlling aspects of the database's operation.

I've applied this PRAGMA settings in the codes for normal and batch operations:

Code link for normal operation:- https://github.com/MonikaK2409/SQLite/blob/main/Task-07/operations.py

Code link for Batch operation:- https://github.com/MonikaK2409/SQLite/blob/main/Task-06/batch_operation.py

Here are the PRAGMA statements used in code comparing with the default values, explained:

1. PRAGMA cache_size:

Purpose: Controls the number of database disk pages that SQLite holds in memory at once.

Value Used: 10000 (number of pages).

Default Value: 2000 pages.

Explanation: Increasing the cache size can improve performance by reducing the need to read from disk, as more pages can be kept in memory.

2. PRAGMA temp_store:

Purpose: Determines where temporary tables, indexes, and other temporary storage go.

Value Used: MEMORY.

Default Value: 0 (DEFAULT), where the default is determined by compile-time options.

Explanation: Setting this to MEMORY forces SQLite to store temporary objects in RAM rather than on disk, which can significantly speed up operations that require temporary storage.

3. PRAGMA synchronous:

Purpose: Controls how aggressively SQLite waits for data to be written to disk.

Value Used: NORMAL.

Default Value: FULL.

Explanation: Setting synchronous to NORMAL offers a balance between performance and data safety. It allows some data to be buffered and written to disk less frequently, improving performance at the potential cost of a higher risk of data loss in the event of a crash.

4. PRAGMA journal_mode:

Purpose: Determines the journaling mode for SQLite, which affects how transactions are logged.

Value Used: WAL (Write-Ahead Logging).

Default Value: DELETE.

Explanation: WAL mode allows for better concurrency and can improve write performance because it allows readers to access the database while a write is ongoing. The default DELETE mode creates a rollback journal to handle transactions, which can be slower.

5. PRAGMA page_size:

Purpose: Sets the size of the database pages.

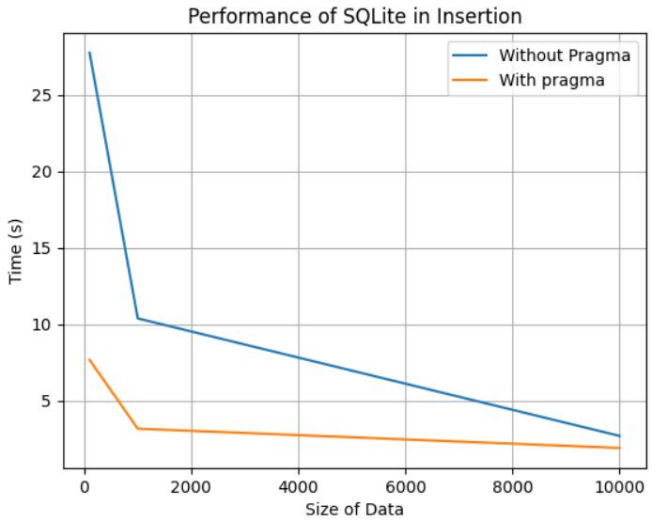
Value Used: 8192 bytes.

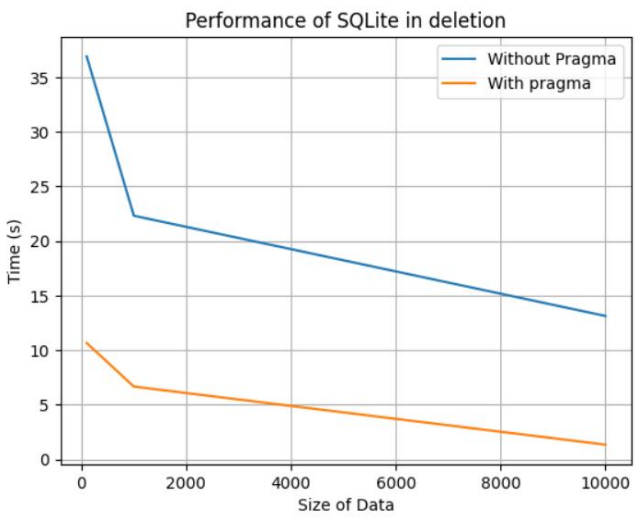
Default Value: 4096 bytes.

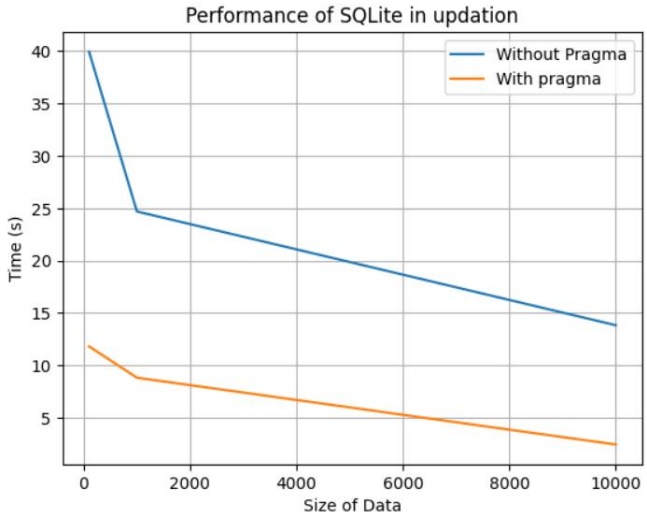
Explanation: Increasing the page size can improve performance for databases with larger records because more data can be read/written in a single operation. However, it might increase the amount of memory used per page in the cache.

<u>The Performance variation before and after using pragma settings:</u>

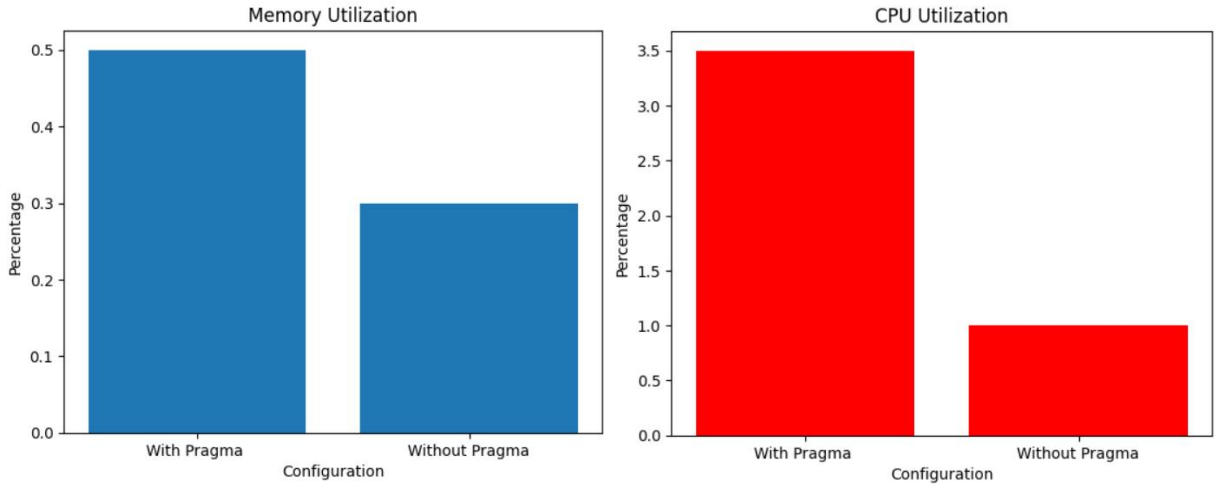
- The below graphs are plotted for 1lakh datasize and time taken for operation is noted.
- Without pragma has default values set as mentioned above. With pragma has the changed pragma values as mentioned above.







- The CPU and memory utilization before and after the configuration change is also monitored.(An average increase in memory and cpu utilization in all operations is aggregated and summarized as below)



Conclusion:

The implementation and configuration of SQLite for managing network flow data demonstrated significant performance improvements through the use of PRAGMA settings. Adjusting the cache size, temporary storage location, synchronization mode, journal mode, and page size allowed for more efficient data handling and optimized database performance. This study highlights the importance of fine-tuning database configurations to achieve optimal results for specific use cases.

Conclusion:

SQLite presents a viable option for managing Network Flow databases due to its lightweight and self-contained architecture. Operating in-process, SQLite avoids the overhead associated with traditional client-server database systems, making it suitable for applications that require efficient and reliable data management, such as network flow analysis. The ability to enhance performance through various PRAGMA options also allows SQLite to handle large datasets and high transaction rates effectively.

However, SQLite does have its limitations. Its performance may not scale as well as more robust, server-based RDBMS when dealing with extremely large volumes of data or high concurrency. Additionally, SQLite's lack of built-in user management and access control can be a disadvantage for more complex security requirements.

Despite these drawbacks, SQLite's widespread use in diverse applications, from web browsers to mobile phones, demonstrates its reliability and flexibility. Its public domain license permits unrestricted use and distribution, making it an accessible choice for developers. The combination of simplicity, efficiency, and the ability to fine-tune performance makes SQLite a practical solution for Network Flow database management, offering a balance between performance, ease of use, and cost-effectiveness.