How does Sqlite3En Work?

Contents:

Problem:

The problem is encryption SQLite databases in Python programming language.

There are two ways to encrypt SQLite databases, either by **encrypting all the values within the database**- this way has several problems- or **encrypt the entire database as a file** by read and encrypted that **file** using the *open* function, then *read* function, then encrypt what was read, then write the encrypted data using the *write* function, this method also has problems

Let's discuss the problems of the two methods:

First method problems

When we encrypt the values within DB, we have two possibilities; either to **deal with the encrypted database with its values as they are(encrypted)** this method is useful when the user (or app) wants to add a new value into the encrypted database, for example, we can take a value from user, encrypts it, and add it in the encrypted database. It is a fast and effective way to add data, but it can't pull data from the encrypted database using the SELECT function, an example:

If the user wants to know the phone number of an employee within the encrypted database, it will use the function

```
SELECT phone FROM db_table WHERE name = 'employee name'
```

So the user asks for the phone number using the name of the employee given, so the database goes to search for the employee's name and then his phone number, but it will not find it because the name given by the user is not encrypted and the name in the database is encrypted, so there is no match between them and the database cannot find it.

You may wonder why we do not encrypt the name of the employee given by the user and then search using the encrypted name, the reason is that when we encrypt a name or a text twice, each time it results in different encrypted values.

That because computer encryption includes several factors, including time and some random values, so it is impossible to match two encrypted values together, even if they are for the same original phrase, and therefore the databases will not be able to find the phone number either.

(There is an encryption method that always gives the same encrypted value, but it is not safe to use).

The second possibility is that we decrypt the entire values of the databases first in the memory where no one can read them and then work on **decrypted** DB directly, but this method will make the program slow at boot (during the decryption process) - if the databases are large in size- ,this way is an appropriate solution in case of small databases only.

Therefore, we did not prefer to use the first method, which is to encrypt the values of the databases

The problems of the second method

Encrypt the entire database as a file using the *open* function and then *read* function then encrypt what is being read and then write the encrypted data using the function *write.*

The problem with this method is that you cannot connect to the resulting encrypted file directly and use it as a database using the *connect* function, because the resulting file is not a database, but an encrypted file, so it must be first decrypted again in order to produce the unencrypted database, then we connect to it using the *connect* function.

After decrypting the file, the unencrypted file must be placed on the hard disk so that we can *connect* to the file and use it with the *connect* function.

We cannot read the file and decrypt it then write it in memory then connect it using *connect* function because the *connect* function (in SQlite3) does not allow us to do that, so the user can find the unencrypted file on the hard disk or hard drive and steal the database.

Solution:

The solution to this dilemma was as follows:

To encrypt the databases, we first connect to the original unencrypted database that we want to encrypt, then create a cursor object, then we convert it into a Python list using the *list* or *cursor.fetchall* function.

So, the database transforms to Python *list* that contain *tuple*, then we convert the last *list* into a **string** using the *str* function, then we encrypt that **string** and write it using the *write* function in a file that ends up with (.Kn)

Decryption

We first create a new and empty database in :memory: using the *connect* function, then we read the (.Kn) file that contains encrypted data using the *read* function, then decipher(decrypt) the string , so we get a non -encrypted string, then convert the non -encrypted string to a new Python list using the *eval* function, then add the resulting list to the database in :memory: that we created a little while ago, so we get a database in :memory: that contains the non -encrypted information .

The user or program can deal with the database in :memory: with ease, as it is not encrypted

Save and encrypted data from the user

When the user or program adjusts the database which is in : memory:, it should be encrypted again and added on the hard disk so that the information will not be lost .

Therefore, after any amendment to the databases within the memory, the app should read the entire database again and convert it into Python list, then convert Python list to string again, then encrypt the sting to (.Kn) file again and write it on the hard disk, so we get a (.Kn) file which contains the database encrypted and modified.

The advantage of this method is relatively quick to decrypt and encryption.

But it has one problem, which is to save the information after its amendment.

In case that the database is large in size, the previous (save) process requires a relatively long time 3-10 seconds, and that is not acceptable in some programs such as sales and accounting programs.

To solve the previous problem, we made some adjustments to the **encryption, decryption** and **save** process.

First encryption of database:

The package (Sqlite3EN) first divides the database (which we want to encrypt)(if it is large in size) into several small databases as each small DB contains 500 elements or row only.

Then encrypt all the resulting small databases to (.Kn)files

Second, **decryption:**

Sqlite3En create a new empty database in :memory: , then Sqlite3En read all the previous .Kn files and decrypt it again and add it to the previous in :memory: database.

Save changing

When the user or program adjusts a value in DB , Sqlite3En knows the modification site(Row modification in :memory: DB) then modify only the .Kn file, which contains that ROW, so we get a speed in the process of saving and modification.

Done, thanks to God.

Yousof Bader Yousof