



[articles](#) [quick answers](#) [discussions](#) [features](#)  
[community](#) [help](#)

Articles / Programming Languages / SQL

Watch



[SQL](#) [.NET](#) [data](#)

# How Much Can Sqlite Handle? Multiple Threads Concurrently Inserting Into Sqlite



raddevus

22 Mar 2024

CPOL

13 min read

5.4K

46

7

14

Rate me:  4.95/5 (4 votes)

An informal study of the Sqlite database and how many concurrent inserts it can handle. Will help you decide if you want to use Sqlite in your projects.

I'm running a sqlite database on my web site backend. I was curious about how much it can handle so I built a little .NET Core console app to exercise a local copy of sqlite with concurrent Inserts using Entity Framework. Along the way, you'll see how easy it is to build an app that uses EF, Sqlite and .NET Core.

[Download cpSqliteStudyV3.zip - 3.1 KB](#)

You can also get the code at my [GitHub repo](#)[^].

## Introduction

I have a web site that uses a simple Sqlite database for the back-end data.

Maybe that sounds odd, but here's what the official Sqlite documentation has to say about that:

## When To Use[^]

Websites SQLite works great as the database engine for most low to medium traffic websites (which is to say, most websites). The amount of web traffic that SQLite can handle depends on how heavily the website uses its database. Generally speaking, any site that gets fewer than 100K hits/day should work fine with SQLite. The 100K hits/day figure is a conservative estimate, not a hard upper bound. SQLite has been demonstrated to work with 10 times that amount of traffic.

The SQLite website (<https://www.sqlite.org/>) uses SQLite itself, of course, and as of this writing (2015) it handles about 400K to 500K HTTP requests per day, about 15-20% of which are dynamic pages touching the database. Dynamic content uses about 200 SQL statements per webpage.

If you've never researched much about Sqlite (and maybe even if you have), those statements may shock you, since most people think of Sqlite as a database for mobile devices.

## What's In It For You

I'm hoping that this article will provide you with the ability to quickly get an idea of whether or not Sqlite is feasible for your own project. Here's what I'll provide:

1. Tutorial / walk-through showing you how to build a quick .NET Core console app.
2. Some helpful explanation of using the "dotnet" command line (various commands which will help you learn to build and support apps built with dotnet.
3. Small Console app you can alter for your purposes
4. A light introduction to C# threading via generating worker threads which will insert concurrently into Sqlite
5. Data which can be examined to discover / decide if Sqlite can be trusted / used.

## Background

However, I was still curious about what would happen with a lot of concurrent requests.

What happens if there are a "large" number of concurrent inserts to the database while another user is attempting to read from it?

That's what this article (lightly) investigates. I say lightly because we will create an entire Console program to run some concurrent inserts against a local Sqlite database, but I'd really like to hear input on what the results mean. I'll provide a way to see some interesting data and hopefully I'll hear back from someone who has some input on what the results mean.

Let's get started.

# Let's Build the Console App Together

I'm using Visual Studio Code with an installation of .NET Core 8.0.202 (SDK) and the 8.0.3 of the .NET Runtime.

## Discovering Your .NET Versions

If you have .NET Core installed and you'd like to see what versions you have, then just open up a terminal window and run:

1. BAT

```
$ dotnet --list-sdks
```

2. BAT

```
$ dotnet --list-runtimes
```

You'll see some output which will give you an idea of what you are running.

Here's what mine looks like:

```
raddev@raddesk:~$ dotnet --list-sdks
6.0.420 [/usr/share/dotnet/sdk]
7.0.407 [/usr/share/dotnet/sdk]
8.0.202 [/usr/share/dotnet/sdk]
raddev@raddesk:~$ dotnet --list-runtimes
Microsoft.AspNetCore.App 6.0.28 [/usr/share/dotnet/shared/Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 7.0.17 [/usr/share/dotnet/shared/Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 8.0.3 [/usr/share/dotnet/shared/Microsoft.AspNetCore.App]
Microsoft.NETCore.App 6.0.28 [/usr/share/dotnet/shared/Microsoft.NETCore.App]
Microsoft.NETCore.App 7.0.17 [/usr/share/dotnet/shared/Microsoft.NETCore.App]
Microsoft.NETCore.App 8.0.3 [/usr/share/dotnet/shared/Microsoft.NETCore.App]
raddev@raddesk:~$
```

## Create New Project (Using Top Level Statements)

Since this is a simple little app, we will create it using the basic Console app and [Top Level Statements](#)[^].

I run Ubuntu 22.04.3 LTS exclusively on my home desktop and I also run a Mac Pro (M3).

I only run Windows remotely (for work) and locally in VirtualBox, but the following commands will allow you to create a console app on any of those three platforms.

Open a terminal window and go to a folder where you want to create the project.

A separate project folder will be created to contain all the files in the project.

BAT



```
$ dotnet new console -o sqliteThreads
```

You will now have a new folder named *sqliteThreads* that contains the basic console app.

## Run a Quick Test

You can run the program real quick to make sure that your .NET Core installation is set up properly.

First, change directory to the new project folder:

1. BAT



```
$ cd sqliteThreads
```

2. BAT



```
$ dotnet run
```

That second command will build the app and run it and you should see the basic "Hello, World!" printed to the console window.

## Add Entity Framework Via Nuget

I decided to use Entity Framework to make all of this a bit quicker so that's the first thing we'll add to the project.

Go to your project folder in your terminal and run the following command:

BAT



```
$ dotnet add package Microsoft.EntityFrameworkCore.Sqlite
```

Obviously, that adds the library that will allow us to use Entity Framework with Sqlite.

If you're interested in additional details about creating a console app using Entity Framework with Sqlite, you can check the Microsoft article [I referenced to learn to do this](#)<sup>[^]</sup>.

Basically, after you run the command to add that package, then all the necessary packages are downloaded and your *.csproj* file will have the following added to it (to reference the EF sqlite package):

XML



```
<PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="8.0.3" />
```

After I add a nuget reference, I always do a build just to make sure everything is still working properly.

BAT



```
$ dotnet build
```

## Add New Class to Represent Data

Let's add a new class which we'll use to represent the data which we'll write to our table.

We'll need a few properties which will help us examine the data after it is written.

1. **Id** - db incremented value for each row inserted
2. **ThreadId** - **String** that holds the name of the thread that is inserting into the db.
3. **Created** - **DateTime** so you can see when the row was inserted

That's all we really need since we're just trying to write a lot of data to Sqlite.

I'll just call it **ThreadData** and I'll put it in a **namespace** named **sqliteThreads.Model** because I created a new folder under my project named **Model**. Later, you will see that in our *Program.cs* file, we will need to add a **using** statement like: **using sqliteThreads.Model** and it'll look like this:

C#



```
namespace sqliteThreads.Model;
class ThreadData{
    Int64 Id{get;set;}
    String ThreadId{get;set;}
    DateTime Created{get;set;}
}
```

Whenever I add a new class or code, I go ahead and build so let's do that again.

BAT



```
$ dotnet build
```

When you do that, you'll probably get a warning that the **ThreadId** "must contain a non-null value when exiting the constructor". It's just trying to let you know that you need to initialize your values before you use them. That will be taken care of later.

## Construct a New ThreadData Object with Initialization

Let's flip over to our *Program.cs* file and add construct a **new ThreadData** object.

Add the following code to the *Program.cs* file.

(You can remove the original line which wrote out the "Hello, World!" statement if you want, or you can leave it.)

C#



```
using sqliteThreads.Model;

ThreadData td = new ThreadData{ThreadId="Main", Created=DateTime.Now};

Console.WriteLine($"Id: {td.Id}, ThreadId:{td.ThreadId}, Created:{td.Created}");
```

When you run that, you'll see something like the following:



```
Id: 0, ThreadId:Main, Created:3/20/2024 1:53:52 PM
```

Now, we know we have some basic data that we can write to our Sqlite db.

## Entity Framework Context Class

To access the Sqlite db and write records, we are using Entity Framework so now we need to add a **DbContext** class.

I basically copied the code from the following Microsoft tutorial and altered it for my purposes: [Getting Started With EF Core\[^\]](#)

I created the **DbContext** class (*ThreadDataContext.cs*) and added it to the *Model* folder.

### Fewer Installs: Didn't Use dotnet-ef Tool

However, I didn't want to make you install the **dotnet-ef** tool (as required by the article linked above to create your database) so I decided to add a direct reference to the Sqlite libraries via Nuget (using **Microsoft.Data.Sqlite**) so we can simply create the database ourselves if it doesn't exist. All of this work is done in the **ThreadDataContext** constructor so there are no worries about creating the database.

### Checks for Existence of Sqlite DB File

When you run the code, the context class will check for the existence of the *thread.db* file and if it doesn't exist, it will

1. create the database file

## 2. add the `ThreadData` table to the database

Here's a snapshot of the `ThreadDataContext` class which should make it fairly clear how the Sqlite database is created. (Keep in mind, a sqlite database is just a file.)

C#

Shrink ▲ 

```
namespace sqliteThreads.Model;

using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using Microsoft.Data.Sqlite;

public class ThreadDataContext : DbContext
{
    // The variable name must match the name of the table.
    public DbSet<threaddata> ThreadData { get; set; }

    public string DbPath { get; }

    public ThreadDataContext()
    {
        var folder = Environment.SpecialFolder.LocalApplicationData;
        var path = Environment.GetFolderPath(folder);
        DbPath = System.IO.Path.Join(path, "thread.db");
        Console.WriteLine(DbPath);


        SqliteConnection connection = new SqliteConnection($"Data Source={DbPath}");
        // ##### FYI THE DB is created when it is OPENED #####
        connection.Open();
        SqliteCommand command = connection.CreateCommand();
        FileInfo fi = new FileInfo(DbPath);
        // check to see if db file is 0 length, if so, it needs to have table added
        if (fi.Length == 0){
            foreach (String tableCreate in allTableCreation){
                command.CommandText = tableCreate;
                command.ExecuteNonQuery();
            }
        }
    }

    // configures the database for use by EF
    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseSqlite($"Data Source={DbPath}");
    protected String [] allTableCreation = {
        @"CREATE TABLE ThreadData
        (
            [ID] INTEGER NOT NULL PRIMARY KEY,
            [ThreadId] NVARCHAR(30) NOT NULL check(length(ThreadId) <= 30),
            [Created] NVARCHAR(30) default (datetime('now','localtime'))
                check(length(Created) <= 30)
        )"
    };
}
```

## Add Some Code to Program.cs & Try It

Now, let's add just a bit of code to our main program and try it out.


We just need to add one line of code in our *Program.cs* which will instantiate the `ThreadDataContext` class and that will create the database file.

```
C#   
ThreadDataContext tdc = new ThreadDataContext();
```

That's it! Now, let's run it.

```
BAT   
$ dotnet run
```

You should see something like the following:

```
  
Id: 0, ThreadId:Main, Created:3/20/2024 2:57:35 PM  
/Users/<redacted-user-name>/Library/Application Support/thread.db
```

**Note:** I'm running this on my Mac PowerBook now so your path may be different.

## You Do Have Sqlite, Don't You?

Now that we've created the database and added the `ThreadData` table, we can examine it with `sqlite3` app.

**Question:** Do you have `sqlite` installed on your machine already?

**Answer:** If you are running Linux or macOS, then it is very likely you do. However, if you don't, you may need to [get it from the sqlite web site here](#)<sup>[^]</sup>. You just need the `sqlite3` executable and maybe there are a couple of DLLs which are included so you probably want the name which is named like: `sqlite-tools-win-x64-3450200.zip` (contains command line tools and command line shell app).

Once you have run our app above, you'll have a path and you'll go to a terminal and run the `sqlite3` command to open the database:

```
  
$ sqlite3 "/Users/<redacted-user-name>/Library/Application Support/thread.db"
```



**Notice** that my path has spaces in it so I am forced to add double-quotes around the full path to the db file.

Once sqlite3 starts, you will see a command-line interface:

```
SQLite version 3.43.2 2023-10-10 13:08:14
Enter ".help" for usage hints.
sqlite> █
```

Go ahead and type `.schema` <ENTER> (notice the leading dot in front of that command):

```
[sqlite> .schema
CREATE TABLE ThreadData
(
  [ID] INTEGER NOT NULL PRIMARY KEY,
  [ThreadId] NVARCHAR(30) NOT NULL check(length(ThreadId) <= 30),
  [Created] NVARCHAR(30) default (datetime('now','localtime')) check(l
ength(Created) <= 30)
);
sqlite> █
```

This allows you to see the tables which are in the database.

Next, you can do a `select`, but there are no records in the DB yet.

SQL



```
select * from ThreadData;
```

Finally, to exit, just type `.exit` (again notice the leading dot in the command).

Now, let's go see about adding a record to our database.

## Adding Data To ThreadData Database

Open up your `Program.cs` file in your dev environment (you are using Visual Studio Code, aren't you?) and add the following code:

C#



```
tdc.Add(td);

tdc.SaveChanges();

Console.WriteLine("Wrote to db");
```

We had already created a `ThreadData` object earlier in our code named `td` so now we simply use our `ThreadDataContext` class to `Add()` the record to our database and then `SaveChanges()` to write the data. This is why I decided to use Entity Framework for this little app. It's just so easy.

Run that code and each time you do, a new record will be added to the database.

I'll let you work out how you can use `sqlite3` to connect to the database and run your select again.

Here's the data that I see in mine after running the app a few times.

```
[sqlite> select * from threaddata;
1|Main|2024-03-20 14:42:36.549199
2|Main|2024-03-20 14:44:57.822971
3|Main|2024-03-20 15:21:45.643168
sqlite> █
```

Now, we can do the fun part. Let's gen up some threads which will all write to the database concurrently. Since we've provided a way to add a `ThreadId`, it means we'll be able to see which thread was doing the work.

## Making Our App Insert Data on Different Threads

It's extremely easy to create a new thread in our app.

We can literally just gen up a `new Thread` like:

C#



```
Thread t = new Thread(() => Console.WriteLine("I'm on a separate thread!"));
t.Start();
```

As a test, go ahead and add those two lines to the top of your `Program.cs` and you'll have a new thread of execution in your program.

The first line creates the thread and the second line starts it running.

According to the fantastic book, [C# 12 In A Nutshell](#)<sup>[^]</sup>, creating a thread this way creates a foreground thread:

### C# 12 In a Nutshell

"By default, threads you create explicitly are foreground threads. Foreground threads keep the application alive for as long as any one of them is running, whereas background threads do not."

Yes, that means if the thread you generated did not complete, then your app will not close.

Run the app again and you'll see something like the following:



```
Hello, World!  
  
Id: 0, ThreadId:Main, Created:3/20/2024 3:31:28 PM  
  
/Users/<redacted-username>/Library/Application Support/thread.db  
  
Wrote to db  
  
I'm on a separate thread!
```

Our code runs an anonymous function (lambda) but we want to be able to pass in the name of our `threadId`, so we'll generate threads which will run a specific function. Let's write that function now.

## WriteData Function For Use With Multiple Threads

I learned quite a few pedantic details while I was writing and testing this method so I'll show it to you here and then hit the high (and low) points to explain the details that may not be obvious.

C#



```
void WriteData(string threadId){  
    ThreadDataContext db = new ThreadDataContext();  
  
    for (int i = 0; i < INSERT_COUNT;i++){  
        try{  
            ThreadData td = new ThreadData{ThreadId=threadId, Created=DateTime.Now};  
            db.Add(td);  
            db.SaveChanges();  
        }  
        catch(Exception ex){  
            Console.WriteLine($"Error: {threadId} => {ex.InnerException.Message}");  
            continue;  
        }  
    }  
}
```

There are a number of points to this function and I'll list some of them which may stand out to you, then I'll try to explain why I've included them.

1. I make each thread do all of their work within the loop in the function. Each thread will stay alive for as long as the `for` loop gate is unsatisfied.
2. `INSERT_COUNT` is a `const int` which allows you to set (at the top of *Program.cs*) the number of inserts that each thread will execute.

3. I **catch** any failure during database **insert** and then just **continue** the loop. Generally, this will be caused by the database being so heavily used that it is locked at the exact moment that this thread is attempting to **insert**. I discovered that even with 13 threads running on my 12 processor machine, this happened very rarely.

There was a little more work to clean up *Program.cs* but here's what it looks like if you want to run it on 13 threads like I did.

C#

Shrink ▲ 

```
using sqliteThreads.Model;

const int INSERT_COUNT = 100;
int insert_count = 0;

// If user passes number of records as valid integer
// then each thread will insert that number of records
// otherwise program will use INSERT_COUNT
if (args.Length > 0){
    try{
        insert_count = Int32.Parse(args[0]);
    }
    catch{
        insert_count = INSERT_COUNT;
    }
}
else{
    insert_count = INSERT_COUNT;
}

Console.WriteLine($"#### Inserting {insert_count} records for each thread. ####");
Thread t = new Thread(() => WriteData("T1"));
Thread t2 = new Thread(() => WriteData("T2"));
Thread t3 = new Thread(()=>WriteData("T3"));
Thread t4 = new Thread(()=>WriteData("T4"));
Thread t5 = new Thread(()=>WriteData("T5"));
Thread t6 = new Thread(()=>WriteData("T6"));
Thread t7 = new Thread(()=>WriteData("T7"));
Thread t8 = new Thread(()=>WriteData("T8"));
Thread t9 = new Thread(()=>WriteData("T9"));
Thread t10 = new Thread(()=>WriteData("T10"));
Thread t11 = new Thread(()=>WriteData("T11"));
Thread t12 = new Thread(()=>WriteData("T12"));

t.Start();
t2.Start();
t3.Start();
t4.Start();
t5.Start();
t6.Start();

t7.Start();
t8.Start();
t9.Start();
t10.Start();
```

```

t11.Start();
t12.Start();

WriteData("Main");

void WriteData(string threadId){
    ThreadDataContext db = new ThreadDataContext();
    var beginTime = DateTime.Now;
    for (int i = 0; i < insert_count;i++){
        try{
            ThreadData td = new ThreadData{ThreadId=threadId, Created=DateTime.Now};
            db.Add(td);
            db.SaveChanges();
        }
        catch(Exception ex){
            Console.WriteLine($"Error: {threadId} => {ex.InnerException.Message}");
            continue;
        }
    }
    Console.WriteLine($"{{threadId}}: Completed - {DateTime.Now - beginTime}");
}
}

```

## Running the Code

Get the code and try it out. I think you'll be amazed.

If you run it as is, you will find that you get 1300 records inserted into the `ThreadData` table.

### Edit: Use Command-Line To Set Insert Count

I altered the code and added the ability for the user to set the `insert_count` on the command line.

Now, you can start the app and set the number of records you'd like each thread to insert like the following:



```

$ dotnet run <number_of_records>
$ dotnet run 1000

```

### Default Value

If you do not provide a value, then it will run 100 inserts per thread.

On my Linux (Ubuntu 22.04.3 LTS) box running an AMD® Ryzen 5 2600x six-core processor × 12, I rarely see an exception thrown which looks like:



```
Error: T10 => $SQLite Error 5: 'database is locked'.
```

I also ran it on my MacBook Pro with 36GB of ram and M3 and:

- 12-core CPU with 6 performance cores and 6 efficiency cores
- 18-core GPU
- 16-core Neural Engine

I don't see any locks at all, which is interesting.

Since Sqlite is really a file database, it could be that all of the speed is based upon the speed of the disk / storage device and caching that is associated. I'm not sure.

## Even With Error, No Data Loss

But, even when I see that error occur, I've discovered that I do not lose any of the inserts.

I need to think about that more, but it is quite amazing.

## Added Some Basic Timing Data

Now, in version 2, when each thread starts, it grabs the `beginTime` and when the thread completes the work in the `for` loop, it calculates the amount of time that it took for it to do its work. It's very basic, but gives you an idea of how long it takes.

## Helpful Queries to Examine Your Data

Try these helpful queries in Sqlite, so you can examine your data:

SQL



```
// each run should insert 1300 records
select count(*) from threaddata;

// Get counts grouped by threadId so you can tell if each thread
// inserted the proper number of times
select threadId, count(*) from threaddata group by threadId;

// take a look at the data and see that each thread does work
// until it gets context switched and
// another thread starts inserting.
select * from threaddata;
```

Here's what the result from the 2nd query above will look like: it displays the number of inserts for each thread.

```
sqlite> select threadid,count(*) from threaddata group by threadid;
Main|100
T1|100
T10|100
T11|100
T12|100
T2|100
T3|100
T4|100
T5|100
T6|100
T7|100
T8|100
T9|100
sqlite>
```

## Conclusion

I believe if you check out this code, you'll be quite impressed with Sqlite. Additionally, its ease of use may encourage you to start using it in your own projects.

Let me know what you think.

## History

- 22nd March, 2024: Fixed minor bug that occurred when multiple threads tried to create the database the first time. Previously it would throw exception and end program and user would have to start program again. Also, fixed erroneous query in article that used order by which was supposed to be group by.
- 20<sup>th</sup> March, 2024: First publication of article and code

## License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOLE\)](#)

Written By

**raddevus**

Software Developer (Senior) RADDev Publishing

 United States

"Everything should be made as simple as possible, but not simpler."

# Comments and Discussions

Add a Comment or Question ?   🔍

Spacing **Relaxed** ▾ Layout **Normal** ▾ Per page **25** ▾

First Prev Next

<b>Write ahead log</b>	<b>TomBed</b>	<b>24-Mar-24 18:47</b>
Re: Write ahead log	raddevus	24-Mar-24 23:47
<b>My vote of 5</b>	<b>Ştefan-Mihai MOGA</b>	<b>23-Mar-24 17:52</b>
Re: My vote of 5	raddevus	23-Mar-24 22:58
<b>Critical Sections?</b>	<b>Roland M Smith</b>	<b>22-Mar-24 2:07</b>
Re: Critical Sections?	raddevus	22-Mar-24 2:35
<b>How many rows per second?</b>	<b>Espen Harlinn</b>	<b>21-Mar-24 8:40</b>
Re: How many rows per second?	raddevus	21-Mar-24 9:20
Re: How many rows per second?	Espen Harlinn	22-Mar-24 9:21
Re: How many rows per second?	Jan Gybas 2024	18hrs 31mins ago
Re: How many rows per second?	raddevus	10hrs 28mins ago
<b>broken images</b>	<b>Graeme_Grant</b>	<b>21-Mar-24 7:06</b>
Re: broken images	raddevus	21-Mar-24 9:20
Re: broken images	raddevus	21-Mar-24 9:36

Last Visit: 24-Mar-24 19:11 Last Update: 26-Mar-24 8:41

Refresh 1



Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

---

[Permalink](#)

[Advertise](#)

[Privacy](#)

[Cookies](#)

[Terms of Use](#)

Layout: [fixed](#) | [fluid](#)

Posted 21 Mar 2024

Article Copyright 2024 by raddevus

Everything else Copyright ©

[CodeProject](#), 1999-2024

Web01 2.8:2024-01-30:1