SQlite

SQlite is an in-process SQL database written in C. SQlite differs from other databases in that it uses a simple file for all storage and it is serverless. This means that if you want to use SQlite through Haskell you need to directly call into c code, using an interface called foreign function interface (FFI). All Haskell libraries which provide these c bindings (like direct-sqlite, persistent..) come with a c file, called The SQLite Amalgamation, which is a concatenation of all SQlite c code into a huge 220.000 lines file. The interaction with foreign functions is notorious for its complications. In this blog we will discuss about proper ways to use SQlite in multithreaded applications and propose a new way, which we have implemented and tested with quickcheck-state-machine (more on this on another blog).

First let's get a bit familiar with SQlite. The SQlite project provides a simple, easy to install, command-line program named sqlite3.

Let's open an sqlite3 db by

sqlite3 sqlite3.db

and let's create a tabe to have something to work on

> create table Person (name Text PRIMARY KEY, age Integer);
> insert into person (Nick, 25);

Databases use transaction to group together operations. In SQlite this is done by wrapping them in begin; .. end; commands.

> begin;
> insert into person (Nick, 25);
> insert into person (John, 20);
> end;

This will fail because Nick is already here and name is a Primary Key. John will not be inserted either.
Now let's try to open a transaction within a transaction

> begin;
> begin;
Error: cannot start a transaction within a transaction
> end;

ok we can't. By default SQlite is always in auto-commit mode. This means that for each command, SQLite starts, processes, and commits the transaction automatically. A begin command makes an sqlite connection leave from autocommit mode. This is important because we don't want to commit each command separately, but group commands together. However trying to leave autocommit mode 2 times is a mistake. It may seem strange to even try to do this, but we later see that this is an important issue when two threads share the same db connection (this is something we can't test from sqlite3).

But what about a second connection? How does it interact with the existing connection?
Lets use sqlite3 again to open a second connection

```
                    # sqlite3 sqlite3.db
                    > .schema
                    CREATE TABLE Person (name Text PRIMARY KEY, age Integer);

> begin;
                    > begin;
```

No error...

```
> insert into person(name,age) values("john", 42);
                    > insert into person(name,age) values("mark", 17);
                    Error: database is locked
```

This means that calling insert is what actually uses the internal SQlite locks and not begin. Since insert is inside a transaction it will continue to hold the lock. When the second transaction tries to insert it finds the db locked.

```
> end;
                    > insert into person(name,age) values("mark", 17);
                    > end;
```

There is a strict version of begin, which takes the locks immediately:

```
> begin immediate;
                    > begin immediate;
                    Error: database is locked
```

as expected, this throws an exception even before trying to insert.

While the db is still busy, let's try something else:

```
# lsof sqlite3.db
COMMAND   PID   USER   FD   TYPE DEVICE SIZE/OFF    NODE NAME
sqlite3 24657 kostas    3ur  REG    8,7     1024 4725038 sqlite3.db
sqlite3 26082 kostas    3ur  REG    8,7     1024 4725038 sqlite3.db
# kill -9 24657

> Killed
                    > begin imediate;
```

The db is unlocked for other connections.

Persistent

Persistent is a Haskell datastore, which provides support for many databases, including SQlite. For an introduction into persistent this https://www.yesodweb.com/book/persistent is great. We will try to get the same behavior as above from persistent.
Here we create our db, and let 2 threads write to the same db connection concurrently:

```
main :: IO ()
main = do
   conn <- open "sqlite.db"
   (backend :: SqlBackend) <- wrapConnection conn (\_ _ _ _ -> return ())
   flip runSqlPersistM backend $ do
```

```
      --  create table
      runMigration $ migrate entityDefs $ entityDef (Nothing :: Maybe Person)
   p0 <- async $ thread0 backend
   p1 <- async $ thread1 backend
   _ <- waitBoth p0 p1
   close' backend

thread0 :: SqlBackend -> IO ()
thread0" backend = forM_ [1..1000] $ \n -> do
   runStdoutLoggingT $ flip runSqlConn backend $ do
      insert_ $ Person ("kostas-" ++ show n) n

thread1 :: SqlBackend -> IO ()
thread1" backend = forM_ [1..1000] $ \n -> do
   runStdoutLoggingT $ flip runSqlConn backend $ do
      insert_ $ Person ("john-" ++ show n) n
```

When we run it we get this error message:
SQLite3 returned ErrorError while attempting to perform step: cannot start a transaction within a transaction

Exactly what we got when we tried two `begins` from the same connection from sqlite3. Same conclusion: we can't run 2 transactions using the same connection. Let's try now to read:

```
thread0 :: SqlBackend -> IO ()
thread0 backend = forM_ [1..1000] $ \n -> do
   runStdoutLoggingT $ flip runSqlConn backend $ do
      (pers :: [Entity Person]) <- selectList [] []
      return ()

thread1 :: SqlBackend -> IO ()
thread1 backend = forM_ [1..1000] $ \n -> do
   runStdoutLoggingT $ flip runSqlConn backend $ do
      (pers :: [Entity Person]) <- selectList [] []
      return ()
```

SQLite3 returned ErrorError while attempting to perform step: cannot start a transaction within a transaction.
Actually Persist wraps in transactions all sql queries, even if they simply read from the database.

It seems this is the wrong approach to use SQlite from multiple thread. We find that persistent provides a way to handle multiple db connections, using Data.Pool.

```
main :: IO ()
main = do
   poolBackend :: Pool SqlBackend <- runStdoutLoggingT $ createSqlitePool "dbs-test/sqlite.db" 2
   runStdoutLoggingT $ flip runSqlPool poolBackend $ do
         runMigration $ migrate entityDefs $ entityDef (Nothing :: Maybe Person)
   p0 <- async $ thread 0 poolBackend
   p1 <- async $ thread 1 poolBackend
   _ <- waitBoth p0 p1
   destroyAllResources poolBackend
```

```
    return ()

thread :: Int -> Pool SqlBackend -> IO ()
thread p backend = forM_ [1..1000] $ \n -> do
    runStdoutLoggingT $ flip runSqlPool backend $ do
        insert_ $ Person (show p ++ "-" ++ show n) n
```

When we run this we soon get
SQLite3 returned ErrorBusy while attempting to perform step: database is locked
which is the same error we noticed at sqlite3. Reads don't get this error, because they don't try to get write locks.

In order to fix this error, persistent provides a retryOnBusy combinator, which wraps a database action and retries it when a ErrorBusy is encountered.

```
thread :: Int -> Pool SqlBackend -> IO ()
thread p backend = forM_ [1..1000] $ \n -> do
    runStdoutLoggingT $ flip runSqlPool backend $ retryOnBusy $ do
        insert_ $ Person (show p ++ "-" ++ show n) n
```

This fixes the error. However if we try to look carefully how these queries are executed, we will see that there is no fairness between the two threads. This happens because in case of BusyErrors, the thread waits on an exponential backoff. So if there is one thread which continuously writes, the other threads starve.

Fairness is one of the issues we tried to solve. Our implementation puts all the write requests on a queue and there is a single long living thread, which takes the requests from the queue and runs them. (this approach is also suggested here https://news.ycombinator.com/item?id=20047918). The write-thread is forked at the beginning of the execution and it waits for db actions on a forever loop. Any thread which wants to write to the db puts their request on a queue, together with a TMVar. Then it waits on the TMVar for the result. If it is an exception it rethrows it. This way it is not visible to the user that it's a different thread which does all the writing.

Our implementation

In our implementaion we model our db actions with this:
```
data AsyncAction r = forall a. AsyncAction (r -> IO a) (TMVar (Either SomeException a))
```

r is the resource (in our case the db connection) that only the write thread has access to.
The TMVar will get the response of the thread. This can be successful or some exception which is wrapped inside SomeException.

User threads can write actions to a queue (we used TBQueue, since they are bounded and can save us from a lot of issues):
```
resp <- newEmptyTMVarIO
writeTBQueue asyncQueue $ AsyncAction action resp
ret <- atomically $ takeTMVar resp
```

Where actions is the db action the user thread wants to execute.

The write thread waits on a loop for new actions:

```
go = do
        AsyncAction action tmvar <- atomically $ readTBQueue queue
        ret <- try $ action resource
        atomically $ putTMVar tmvar ret
        go
```

Our implementation is very similar to async and it was influenced by it a lot and in fact we initially tried to use it instead. The difference is that in async the thread lives only for the duration of the execution of the IO action and cannot return the result without dying. In our case we want a thread which continues to live. Another package called immortal provides similar functionality, but it is more general purpose and doesn't directly facilitate us.

As you may have noticed our implementation is not limited to SQlite or dbs, but it is built to be a lot more general. We have implemented it as a different package called async-queue and plan to publish on hackage. We have also adopted Persistent to use this package. We believe it can be easily ported to other Haskell datastores which support SQlite.

It may seem that serializing writes on a thread decreases parallelism, but as we saw, SQlite doesn't allow concurrent writes, anyway. Our implementation works very well with the new atomic transaction mode provided by SQlite, called Write-Ahead-Log. https://www.sqlite.org/wal.html, which Persistent enables by default. This mode enables concurrent reads, so serializing only the writes makes perfect sense. Keeping a connection to write open for very long is also a good option. In WAL mode, SQlite manages to limit a lot syncing data to disk. However closing the connection forces an fsync(), which we avoid. fsync() is an operation in which the content of data found in memory buffers or in page tables and caches is forced to the disk and is one of the biggest performance issues for databases.

Foreign Calls

This approach simultaneously solves a different issue. Calling into foreign code has a big disadvantage: The thread cannot be interrupted. This can be frustrating to the user, especially when he tries to stop the execution with a ^C action. This is the reason why it is usually advised not to directly call into foreign code, but to fork a thread and let this thread do the foreign calls.

(from Parallel & Concurrent Programming in Haskell):

```
do
  a <- async $ c_read fd buf size
  r <- wait a
```

This way the user thread waits on a STM for the execution to end. This makes the thread interruptible: a thread waiting on a STM can be easily interrupted by an asynchronous exception.

As a matter of fact, this is what we do here. But instead of forking a thread each time we want to access the db, we fork a thread at the beginning which waits for db actions. Actually, when someone uses System.IO functions he get the same functionality behind the scenes, from the IO Manager. The I/O manager (amongst other things) delegates some IO operations to special threads, so that the user threads are interruptible. It's interesting to see that the structures of I/O managers are similar to the structures we use. The runtime keeps a global list of pending events, called pendingEvents, containing a elements of the following data type (simplified):

```
data IOReq
  = Read   Fd (MVar ())
  | Write  Fd (MVar ())
```

We used TMVar instead of MVars, in order to use the existing STM mechanism that TBQueue uses. This provides better composability, but we may need to take a look at the performance implications (MVars are usually faster).

Bound Threads

There is an additional reason someone may want to have a long living thread. Some foreign libraries require to always be called by the same os thread. OpenGL is usually mentioned as one of these cases. These libraries keep an implicit state, based on the os thread which called them. Calling them by different os thread means that this implicit state gets lost. The reason we keep mentioning os here is because in general Haskell threads do not directly map to os threads. The Haskell runtime manages a limited pool of os worker threads (if -threaded is not specified it will be a single thread) and multiplexes Haskell threads to them. This approach allows extremely lightweight threads, since context-switching requires no os interaction. However this also means that Haskell threads can jump around between different os threads. Haskel provides a mechanism to keep a thread bound to an os thread: forkOS and asyncBound use this functionality. In our implementation we extend further this functionality: we give the option to make the long living thread os bound. By doing so all foreign calls are always executed by the same thread, without any effort from the programmer. We haven't explored using async-queue in these kind of libraries, but we believe it can make them easier to use.

We are not entirely convinced that using OS bound threads is necessary in SQlite, but investigating this issue brought us to a shocking realization. We have noticed that SQlite uses some mutexes called recursive mutexes, which protect multiple threads of the same connection to access critical areas. Each connection, on initiation, also initiates one of these mutexes. These kind of mutexes have the notion of ownership: the os remembers which thread owns them. This allows the same thread to take the same mutex again and again without deadlocking, with the responsibility of the same thread to unlock it the same number of times. SQlite uses these mutexes and also exports them and suggests using them in some cases https://www.sqlite.org/c3ref/errcode.html. We were tempted to try this out...

Let’s create Haskell bindings for entering the mutex of a connection:

```
foreign import ccall "sqlite3_mutex_enter_connection"
  mutex_enterC :: Ptr () -> IO ()
mutex_enter :: Connection -> IO ()
mutex_enter (Connection _ (Connection' database)) = do
    mutex_enterC database

foreign import ccall "sqlite3_mutex_leave_connection"
  mutex_leaveC :: Ptr () -> IO ()
mutex_leave :: Connection -> IO ()
mutex_leave (Connection _ (Connection' database)) =
    mutex_leaveC database
```

Now let’s see how well these mutexes protect a critical area from 2 threads.

```
main :: IO ()
main = do
    db <- open "sqlite3-mutex.db"
    mvar <- newMVar False
    a <- async $ mutexThread 0 db mvar
    b <- async $ mutexThread 1 db mvar
```

```
    _ <- replicateM 10 $ async $ forM_ [1..] $ \n ->
                -- here we simulate the existence of other threads.
                -- These create bigger contention for the same
                -- os threads.
                dummyIOAction
    _ <- waitBoth a b
    return ()

mutexThread :: Int -> Connection -> MVar Bool -> IO ()
mutexThread n db mvar = forM_ [1..1000] $ \n -> do
    _ <- mutex_enter db
    -- critical area.
    modifyMVar_ mvar $ \x -> case x of
       True -> throwIO $ ConcError $ "Thread " ++ show n ++ " expected False on trial " ++ show n
       False -> return True
    dummyIOAction
    modifyMVar_ mvar $ \x -> case x of
       False -> throwIO $ ConcError $ "Thread " ++ show n ++ " expected True on trial " ++ show n
       True -> return False
    -- end of critical area.
    mutex_leave db
    print n

data ConcError = ConcError String deriving (Show, Typeable, Exception)
dummyIOAction :: IO ()
dummyIOAction = writeFile "/dev/null" (show $ fib 42)
```

We have 2 threads trying to run on a critical area and we protect this critical area with these mutexes. Note that we create some additional threads which run dummy actions. This creates bigger contention to the scheduler for the os threads available. In addition we compile with -threaded -rtsopts "-with-rtsopts=-N2", which allows using only 2 os threads. This makes contention even bigger. Note that in this implementation we don't care a lot about exceptions.

If we try to execute this it will sometimes deadlock and sometimes fail, with something like ConcError "Thread 6 expected False on trial 6"
On the other hand, if we replace async with asyncBound, nothing bad happens.

I believe it's pretty clear what happens at this point. If our Haskell thread gets context switched and later scheduled on a different os thread, while owning this mutex, the results are disastrous. We printed the thread ids at the beginning of the critical area and after returning form the dummyIOAction and we confirmed that the threadId changes. Note that we are not referring to the thread Id returned by `Control.Concurrent.myThreadId` but from `System.Posix.Thread.myThreadId`, which is basically a system call for unix like systems and returns the id of the os thread currently running. This thread id can dynamically change in the lifetime of a Haskell thread which is not os bound.

What's interesting is that the program deadlocks even if only a single thread tries to enter the critical area (that is if we completely erase the b thread and replace waitboth a b by wait a). This is because as we mentioned the recursive mutexes remember the os thread which owns them and can't allow other threads to unlock them.

If we compile our executable without -threaded, the Haskell runtime can only manage a single os thread and multiplexes all Haskell threads to it. This means that the mutex here provides no safety since all Haskell thread map to a single os thread and ownership of the mutex means that all Haskell threads own it. The execution fails with:
ConcError "Thread 1 expected False on trial 1"

Deadlock cannot happen in this case. If we have a single thread trying to get in the critical area no error can occur. Also using asyncBound is impossible, since without -threaded, ghc has no way to know how to fork new os threads, so it fails:

user error (RTS doesn't support multiple OS threads (use ghc -threaded when linking))

It may seem very strange that we even use mutexes from Haskell code. But we must not forget that we use them every time we call into SQlite. It seems though that threads own mutexes only during the duration of a foreign calls and don’t return from them while still holding the mutexes. This successfully hides the problem, because during a foreign call, the Haskell thread which initiated it cannot migrate to a different os thread, so the ownership problems we mentioned above no longer exists.

Results

We have implemented a new library which can be used to access SQlite. Our approach provides some important benefits:
- ensures fairness since writes are enqueued
- provides a safe wait to call into foreign functions.
- our implementation take good advantage of the new SQlite WAL mode and minimizes expensive fsync
- our library can have further use cases, like protection from libraries which need os bound threads.

In addition we have adopted our implementation in persistent and we have thoroughly tested it with quickcheck-state-machine (more on this on another blog), even with tens of concurrent threads (we have very recently implemented this functionality in q-s-m https://github.com/advancedtelematic/quickcheck-state-machine/pull/324).

[1] Parallel and Concurrent Programming in Haskell
[2] Marlow, S., Peyton Jones, S., and Thaller, W. (2004). Extending the Haskell Foreign Function Interface with concurrency. In Proceedings of Haskell Workshop, Snowbird, Utah, pages 57–68.
[3] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages, pages 295–308, St Petersburg Beach, Florida, January 1996. ACM.