Asynchronous Database Access with Qt 4.x

Dave Berton

Abstract

How to code around the default synchronous database access in Qt 4.

The database support in Qt 4.x is quite robust. The library includes drivers for Oracle, PostgreSQL, SQLite and many other relational databases. Out of the box, the Qt database library also contains bindings for many widgets and provides data types for the transparent handling of result sets coming from a database. But, your application can pay a price for these conveniences. All database access is synchronous by default, which means that intensive and time-consuming SQL queries normally will lock up the UI unless precautions are taken. Using stored procedures on the server can sometimes help the situation; however, this is not always possible or desirable. And often, the length and cost of the queries generated by your application simply cannot be known in advance, so the door is left open for undesirable UI behavior. People don't want their application to “lock up” at odd moments; however, this is the default behavior, and so we must contend with it.

Fortunately, Qt 4.x also has robust support for multithreaded programming. By placing the heavy-duty database work in separate threads, the UI is free to respond to the user normally, without ungraceful interruptions. As with all concurrent programming, however, you must take precautions to ensure the correct sequence of interactions between threads. For example, when sharing data among threads, guard it properly using mutexes. When communicating between threads, consider carefully how the interaction will behave, and in what sequence. In addition, when utilizing a database connection within a thread separate from the UI thread, you must pay attention to some extra caveats. A proper implementation that keeps certain things in mind will make significant improvements in the UI behavior and responsiveness of a database application.

Thread Strategies

There are several ways to distribute the database load to separate threads of execution. Fortunately, all of them share the same characteristics when it comes to the details of creating and using a database connection properly. The primary consideration is to use a database connection only within the thread that created it. For regular synchronous applications, the default behavior is fine. The QSqlDatabase::addDatabase() static function creates a database connection within the context of the application's main UI thread. Queries executed within this same thread will then cause blocking behavior. This is to be expected.

In order to run queries in parallel with the main UI thread, so that they do not interrupt the main event processing loop, a database connection must be established in the thread in which the query executes, which should be separate from the main UI thread. However you structure the threading in your application, your design must be able to establish a connection within the context of each thread that will be performing database work.

For example, creating a thread pool in which a few threads handle the load of querying the database in a round-robin fashion (without the overhead of creating and destroying threads all the time) will push the time-consuming work outside the main event loop. Or, depending on the needs of your application, you simply can spawn threads on an as-needed basis to perform database work. In either case, you must create a connection per thread.

There is a further limitation (imposed by most of the underlying database-specific libraries used by Qt). As a general rule, connections cannot be shared by multiple threads. This means you cannot simply create a pool of connections on startup and hand them out to various threads as needed. Instead, each thread must establish and maintain its own connection, within its own context. To do otherwise is undefined, and probably disastrous. Multiple separate connections can be established in each thread by using the name parameter of the QSqlDatabase::addDatabase() static function, as shown in Listing 1.

Listing 1. Create two instances of QThread, one for queries, another for updates.*9602l1.qrk*

```
class QueryThread : public QThread
{
  public:
    QueryThread( QObject* parent = 0 )
    {
      //...
    }
    void run()
    {
        QSqlDatabase db = QSqlDatabase::addDatabase(
        ↪"QPSQL", "querythread" );
        // use 'db' here
    }
};

class UpdateThread : public QThread
{
  public:
    UpdateThread( QObject* parent = 0 )
    {
      //...
    }
    void run()
    {
        QSqlDatabase db = QSqlDatabase::addDatabase(
        ↪"QPSQL", "updatethread" );
        // use 'db' here
    }
};
```

In Listing 1, the thread objects establish two different database connections. Each connection is named separately, so that QSqlDatabase can maintain them properly in its internal list. And, most important, each connection is established within the separate thread of execution of each object—the run() method is invoked by QThread::start() once the new thread of execution is launched. The mechanism provided by QSqlDatabase to create new connections is thread-safe. Listing 2 shows another example of a more generic approach. *Garrick, shrink code in listing 2.*

Listing 2. A More Generic Approach with a Single Instance of QThread Used Twice*9602l2.qrk*

```
class QueryThread : public QThread
{
```

```
  public:
    QueryThread( const QString& name )
      : m_connectionname(name)
    {
      //...
    }
    void run()
    {
      QSqlDatabase db =QSqlDatabase::addDatabase( "QPSQL", m_connectioname );
      //...
      db.open();
      forever
      {
        // wait for work
        // and then execute it...
      }
    }
}

void main()
{
  QApplication app(...);
  MainWin mw;
  QueryThread db1("queries");
  db1.start();
  QueryThread db2("updates")
  db2.start();
  //...
  mw.show();
  app.exec();
}
```

The pseudo-code in Listing 2 creates and starts two worker threads. Each thread establishes a named connection to the database and waits for work to do. Each thread always will deal only with its own database connection, which is never shared or visible outside the worker thread object.

Setting up thread-specific connections and running queries within that thread is only the first part of the problem. A decision needs to be made regarding how data is shuffled back and forth between the worker threads and the main application UI thread. Additional methods will give the thread object some database work to perform, and those methods will themselves need to be thread-safe.

Moving Data between Threads

You should observe all of the usual caveats surrounding the sharing of data between different threads of execution, including the proper uses of mutexes and wait conditions. In addition, there is an added complication regarding the size of the data, which potentially can be extremely large in the case of result sets returned from the database.

Qt provides several specialized mechanisms for sending data between threads. You can post events manually to any object in any thread using the thread-safe function QCoreApplication::postEvent(). The events will be dispatched automatically by the event loop of the thread where the destination object was created. To create an event loop within each thread, use QThread::exec(). Using this method, threads are given “work” to do in the form of events. QCoreApplication passes these events in a thread-safe manner from the application thread to the worker thread, and they will be handled by the worker thread within its own execution context. This is critical, because the worker thread will be utilizing its database connection only from within its own context (Listing 3).

Listing 3. Code That Shares Information from Worker Thread to Application Thread via Event*9602l3.qrk*

```
class WorkEvent : public QEvent
{
  public:
    enum { Type = User + 1 }
    WorkEvent( const QString& query )
      : QEvent(Type)
      , m_query(query)
    {}

    QString query() const
    {
      return m_query;
    }
  private:
    QString m_query;
};

QueryThread thread;
thread.start();
//...
WorkEvent* e = new WorkEvent("select salary from employee
 ↪where name='magdalena';");
  app.postEvent( &thread, e );
  //...
```

This method works in the other direction as well. Events posted by individual worker threads will show up back in the main UI event loop for handling, within the context of the UI's thread of execution. These events can, for example, contain the result of a query or database update. This approach is convenient, as the worker threads can simply “post it and forget it”, and Qt will take care of the inter-thread communication, mutexing and memory management for you.

Constructing all the necessary events and event handlers for this type of system has advantages—most notably compile-time type checking. However, getting all the events designed and dealt with properly can be an onerous task, especially if your application has multiple types of database queries to perform, with multiple return types, each of which needing an associated event and event handler and so forth.

Fortunately, Qt permits signals and slots to be connected across threads—as long as the threads are running their own event loops. This is a much cleaner method of communication compared to sending and receiving events, because it avoids all the bookkeeping and intermediate QEvent-derived classes that become necessary in any nontrivial application. *Communicating between threads now becomes a matter of connecting signals from one thread to the slots in another, and the mutexing and thread-safety issues of exchanging data between threads are handled by Qt.*

Why is it necessary to run an event loop within each thread to which you want to connect signals? The reason has to do with the inter-thread communication mechanism used by Qt when connecting signals from one thread to the slot of another thread. When such a connection is made, it is referred to as a queued connection. When signals are emitted through a queued connection, the slot is invoked the next time the destination object's event loop is executed. If the slot had instead been invoked directly by a signal from another thread, that slot would execute in the same context as the calling thread. Normally, this is not what you want (and especially not what you want if you are using a database connection, as the database connection can be used only by the thread that created it). The queued connection properly dispatches the signal to the thread object and invokes its slot in its own context by piggy-backing on the

event system. This is precisely what we want for inter-thread communication in which some of the threads are handling database connections. The Qt signal/slot mechanism is at root an implementation of the inter-thread event-passing scheme outlined above, but with a much cleaner and easier-to-use interface.

For example, you can create two simple connections between the main UI object and a worker thread object; one to add a query to the worker thread and another to report back the results. This simple setup, only a few lines of code, establishes the main communication mechanism for an asynchronous database application (Listing 4). *Garrick, shrink font in listing 4.*

Listing 4. Sharing information across threads is cleaner with signals and slots.*9602l4.qrk*

```
class Worker; // forward decl
class QueryThread : public QThread
{
  QueryThread();
  signals:
    void queryFinished( const QList<QSqlRecord>& records );
  slots:
    void slotExecQuery( const QString& query );
  signals:
    void queue( const QString& query );
  private:
    Worker* m_worker;
};

int main()
{
  QApplication app;

  MainWin mw;

  QueryThread t;
  t.start();

  connect(&mw, SIGNAL( execQuery(const QString&)),
          &t, SLOT( slotExecQuery(const QString&)));

  connect(&t, SIGNAL( queryFinished(const QList<QSqlRecord>&)),
          &mw, SLOT( slotDisplayResults(const QList<QSqlRecord>&)));

  mw.show();
  return app.exec();
}
```

Here, the MainWin and the QueryThread objects communicate directly with one another via signals and slots in the usual way. The trick here is that the QueryThread object, behind the scenes, utilizes a Worker object to perform all the work. Why is this extra level of indirection necessary? Because we want to dispatch the work to an object that is associated with a completely separate thread of execution. Notice above that QueryThread is instantiated and start()ed within the main thread of execution; therefore, QueryThread will “belong” to the application's main thread. Connecting signals from the application's widgets to its slots will be via “direct” connections—they will be invoked immediately by Qt, in the usual way, and thus be blocking, synchronous function calls. Instead, we are interested in “pushing” the signals into slots that are running in a separate thread of execution entirely, and this is where the internal Worker class comes into play (Listing 5). *Garrick, shrink font in listing 5.*

Listing 5. An extra level of indirection makes execution more asynchronous.*9602l5.qrk*

```
class Worker : public QObject
{
  Q_OBJECT

   public:
    Worker( QObject* parent = 0);
    ~Worker();

  public slots:
    void slotExecute( const QString& query );

  signals:
    void results( const QList<QSqlRecord>& records );

   private:
     QSqlDatabase m_database;
};

void QueryThread::run()
{
  // Create worker object within the context of the new thread
  m_worker = new Worker();

  // forward to the worker: a 'queued connection'!
  connect( this, SIGNAL( queue( const QString& ) ),
           m_worker, SLOT( slotExecute( const QString& ) ) );
  // forward a signal back out
  connect( m_worker, SIGNAL( results( const QList<QSqlRecord>& ) ),
           this, SIGNAL( queryFinished( const QList<QSqlRecord>& ) ) );

  exec();  // start our own event loop
}

void QueryThread::execute( const QString& query )
{
  emit queue( query ); // queues to worker
}
```

Utilizing this Worker class internally, QueryThread can dispatch SQL queries properly to a separate thread by employing the convenient inter-thread signal/slot queued connections provided by Qt. QueryThread encapsulates the idea of a thread, by being derived from QThread and being able to start() a new thread of execution, and it also encapsulates the idea of a worker (by exposing convenient methods that perform database work, which are internally dispatched to a separate thread of execution). So, instead of tangling with all the necessary events and event handlers to accomplish the same task, the Qt metaobject system rigs up all the necessary code for you and exposes the connect() function to trigger it all. Signals can be emitted at any time and will be dispatched to the destination object correctly, within that object's context.

It is important to note that, above, the execute() method of QueryThread is intended to be invoked synchronously by the main application. QueryThread presents this method not only as a convenience; indeed, it also encapsulates and hides all the queued connection details from the user of the QueryThread class. When execute() is invoked, QueryThread simply emits it as a signal to the worker. Because the worker is an object “living” in a separate thread, Qt will “queue” the connection and pass it through the event loop so that it arrives in the proper execution context of the worker—essential so the worker can

utilize its database connection according to the rules that are laid out in the beginning of this article. Finally, any signals coming from the worker are “forwarded” back out through the QueryThread interface—another convenience for the users of QueryThread, which also serves to hide all the details of the Worker class from those who really don't need to know about it in the first place.

If the sizes of your queries are gigantic, or potentially can be gigantic, you should consider a different strategy for dealing with the result sets. For example, writing the results out to disk before passing them back to the UI thread will reduce the memory load of your application, at the cost of some runtime speed. However, if you are already dealing with massive queries, the speed hit likely will be minor in comparison with the overall execution time of the query. Because all database queries execute in a completely separate thread from the UI, users will not perceive any significant lag. Store the intermediate results in a text file, or for maximum flexibility, in a local SQLite database.

One element of this puzzle still needs to be solved. The queued connection mechanism relies on the thread's event loop to invoke a slot within that thread's context properly, as discussed previously. However, in order to marshal the data necessary to dispatch the event to another thread of execution, the Qt object system must be made aware of any custom data types used in the signal/slot declaration. Failure to register custom data types will cause the queued connection to fail, because in order to dispatch the event, a copy of each of the signal's parameters must be made. Fortunately, it is a simple matter to “register” additional types, as long as those types have public constructors, copy constructors and destructors (Listing 6). *Garrick, shrink font in listing 6.*

Listing 6. You must register any custom data types in order to share them.*9602l6.qrk*

```
// first, make the object system aware
qRegisterMetaType< QList<QSqlRecord> >("QList<QSqlRecord>");
// now set up the queued connection
connect( m_worker, SIGNAL( results( const QList<QSqlRecord>& ) ),
         this, SIGNAL( queryFinished( const QList<QSqlRecord>& ) ) );
```

The sample application provided with this article implements the strategy outlined above, in which queries are executing in parallel with the rest of the application [the application is available for download from the *Linux Journal* FTP site, ftp://ftp.linuxjournal.com/pub/lj/issue158/9602.tgz]. The UI is not disturbed while the queries are underway. A queued connection is created between the QueryThread interface and the encapsulated worker thread after the appropriate types are registered with the Qt metaobject system. This allows the separate threads to communicate safely with one another, with minimal overhead and code complexity. The sample application was tested with SQLite and PostgreSQL; however, it will work with any database connection supported by Qt that enforces the same connection-per-thread limitation.

Summary

The following points should be kept in mind when designing asynchronous database applications with Qt:

- Create a database connection per thread. Use the name parameter of the thread-safe QSqlDatabase::addDatabase() method in order to distinguish various database connections.

- Encapsulate the database connection within worker thread objects as much as possible. Never share a database connection with another thread. Never use a database connection from any thread other than the one that created it.

- Manage communication between threads using the tools provided by Qt. In addition to QMutex, QSemaphore and QWaitCondition, Qt provides much more direct mechanisms: events and signals/slots. The implementation of signals/slots across thread boundaries relies on events; therefore, ensure that your threads start their own event loop using QThread::exec().

- Register unknown types with the Qt metaobject system. Any unknown types cannot be marshaled properly without first invoking qRegisterMetaType(). This enables a queued connection to invoke a slot in a separate thread within that thread's context using new types.

- Utilize queued connections to communicate between the application and the database threads. The queued connection provides all the advantages for dealing with asynchronous database connections, but maintains a simple and familiar interface using QObject::connect().