MongoDB Connection Pooling

This is a slightly modified version of the original blog article posted by Chris Chang of mLab on 11/06/2013.

Connection pools

As your application grows in functionality and/or usage, managing resources becomes increasingly important. Failure to properly utilize connection pooling is one major "gotcha" that we've seen greatly impact MongoDB performance and trip up developers of all levels.

Creating new authenticated connections to the database is expensive. So, instead of creating and destroying connections for each request to the database, you want to re-use existing connections as much as possible. This is where connection pooling comes in.

A *connection pool* is a cache of database connections maintained by your driver so that connections can be re-used when new connections to the database are required. When properly used, connection pools allow you to minimize the frequency and number of new connections to your database.

Connection churn

Used improperly however, or not at all, your application will likely open and close new database connections too often, resulting in what we call "connection churn". In a high-throughput application this can result in a constant flood of new connection requests to your database which will adversely affect the performance of your database and your application.

Opening too many connections

Alternately, although less common, is the problem of creating too many MongoClient objects that are never closed. In this case, instead of churn, you get a steady increase in the number of connections to your database such that you have tens of thousands of connections open when your application could almost certainly do with far fewer. Since each connection takes RAM, you may find yourself wasting a good portion of your memory on connections which will also adversely affect your application's performance.

Although every application is different and the total number of connections to your database will greatly depend on how many client processes or application servers are connected, in our experience, any connection count greater than 1000 - 1500 connections should raise an eyebrow, and most of the time your application will require far fewer than that.

MongoClient and connection pooling

Most MongoDB language drivers implement the MongoClient class which, if used properly, will handle connection pooling for you automatically.

Let's look at a concrete example using the Node.js driver. Creating new connections to the database using the Node.js driver is done like this:

```
const MongoClient = require('mongodb').MongoClient;
const assert = require('assert');

// connection URL
const url = 'mongodb://localhost:27017/dbName';

// Use connect method to connect to the server
MongoClient.connect(url, function(err, db) {
  assert.equal(null, err);
  console.log("Connected successfully to dbName");
  db.close();
});
```

When you create a connection using MongoClient, ***you want to call 'connect' once during your apps initialization phase vs. on each database request***. Let's take a closer look at the difference between doing the right thing vs. doing the wrong thing.

First, consider the following example:

```
let express = require('express');
let MongoClient = require('mongodb').MongoClient;
let app = express();

const MONGODB_URI = 'mongo-uri';

app.get('/', function(req, res) {

  // BAD! Creates a new connection pool for every request

  MongoClient.connect(MONGODB_URI, function(err, db) {
    if (err) throw err;

    db.collection('test').find({}, function(err, docs) {
      docs.each(function(err, doc) {
        if(doc) {
          res.write(JSON.stringify(doc) + ""n");
        } else {
          res.end();
        }
      });
    });
  });
});

app.listen(3000);
console.log('Listening on port 3000');
```

The above example (no pooling) is **bad** for a number of reasons:

- It calls connect() in *every request handler*.
- It establishes new connections for every request (connection churn).
- It initializes the app (app.listen()) before database connections are made.

Contrast the above example to the following:

```
let express = require('express');
let MongoClient = require('mongodb').MongoClient;
let app = express();

const MONGODB_URI = 'mongo-uri';
let db;

// Initialize connection once, reuse the database object

MongoClient.connect(MONGODB_URI, function(err, database) {
  db = database;
  app.listen(3000);
  console.log('Listening on port 3000');
});

app.get('/', function(req, res) {
  db.collection('test').find({}, function(err, docs) {
    docs.each(function(err, doc) {
      if(doc) {
        res.write(JSON.stringify(doc) + ""n");
      } else {
        res.end();
      }
    });
  });
});

app.get('/post', function(req, res) {
  db.collection('test').insert({ docid: Math.random() }, function(err) {
    res.end('Successful Insert!');
  })
});
```

This example (with pooling) is better than the first for the following reasons:

- It calls connect() once.
- It reuses the database variable (reuses existing connections).
- It waits until the database connection is established before it starts listening on the network port.

If you run the first example and refresh your browser enough times, you'll quickly see that your MongoDB has a hard time handling the flood of connections and will terminate.

Further consideration - connection pool size

Most MongoDB drivers support a parameter that sets the max number of connections (pool size) available to your application. The connection pool size can be thought of as the max number of concurrent requests that your driver can service. The default pool size for the node driver is 5. If you anticipate your application receiving many concurrent or long-running requests, we recommend increasing your pool size- adjust accordingly!