Relational vs Document Database

by

Kevin Gilbert

the Faculty of the School of Information Technology
in Partial Fulfillment of the Requirements for
the Degree of Bachelor of Science
in Information Technology

© Copyright Kevin Gilbert

The author grants to the School of Information Technology permission
to reproduce and distribute copies of this document in whole or in part.

_________________________________________  4/17/2014
Kevin Gilbert ___________________________  Date

_________________________________________  4/17/2014
Russell E McMahon, Faculty Advisor ___________________________  Date

University of Cincinnati
College of
Education, Criminal Justice, and Human Services
Table of Contents

List of Figures ......................................................................................................................... 3
Abstract .................................................................................................................................... 4
Introduction ............................................................................................................................ 5
Problem ..................................................................................................................................... 5
Project Description .................................................................................................................. 6
Business Benefit ...................................................................................................................... 8
Solution ..................................................................................................................................... 8
Conclusion ............................................................................................................................... 13
References ............................................................................................................................... 15
List of Figures

Figure 1. Node.js vs C#...........................................................................................................6
Figure 2. Savings Each Year......................................................................................................7
Figure 3. Node.js to SQL recursive function.............................................................................9
Figure 4. Node to SQL. Recursive Function.............................................................................10
Figure 5. Node to MongoDB.................................................................................................10
Figure 6. Final
Timings......................................................................................................................................12
Abstract

Non-relational data in a relational database is a missed opportunity. Due to the nature of relational database schema, there are a lot of wasted connections that can cause slowdowns and lost performance. Document databases are designed to be fast and scalable. Moving away from the restrictions of schema and cross table joining has allowed for increased performance and less upkeep due to no schema changes. The goal of my project is to prove this theory, to increase our knowledge of document databases and to prove they are a viable solution in a business.
Introduction

iSqm believes that using non-relational data is a relational database is causing a problem. Data that is only ever accessed from the front end should be in a database that is consistent with their needs. Due to the nature of relational database schema, there is a lot of wasted space causing slowdowns and lost performance. iSqm believes in using the right tool for the job, therefore I am turning MongoDB to use as a document database to combat our problem and to improve performance, scalability and future proof our data.

Document databases are designed to be fast and scalable. Moving away from the restrictions of schema and cross table joining has allowed these databases to be free from the limitations of the classic relational database design. It will allow us to make less schema changes which will result in less downtime during releases. Having many unused or null columns in relational data is unnecessary when the data can be made into a document database. Also, due to having to navigate through layers of ORMs and DLLs there is a lot of performance loss.

Problem

Right now, we have a department that has a high-turnover rate and involves a very manual process to insert new data for our company. This department is in charge of finding new projects that can be uploaded to the website in order for General Contractors to bid on the projects. As of now, this department has been using a program that was coded in the year 2000 using VBScript and has in-line SQL to do its transactions. All of which is abhorrent by today’s standards. The goal of my project is
to test out different environments with different databases and give a final suggestion on which to use for production development. I believe due to the insert and read needs, along with the fact that this data does not need to appear immediately, lends itself to be perfect for a document database.

MongoDB is an open-source document database that allows many different languages to interact with it. In my case, the front end developers will be using node.js to access the data which is one of the many reasons that Mongo is the DB that was decided upon. While I believe that MongoDB would be an excellent choice, there are other document databases out such as RavenDB which has more support for .Net environments. Document databases also have the idea of eventual consistency, which means as soon as data is entered it will appear for some users, but not others. This is all dependent on how sharded the database is and how long it takes for the data to be spread throughout the database.

**Project Description**

In order to prove out my theory, I have created four separate programs that each have different purposes yet, all lead to the same goal. All of these programs work in the same way. Each one will do Insert, Update, Read and Delete statements and I have captured metrics on how each one performs. The goal of this project is to prove out that with basic data or key-value pairs, a document database is superior. iSqFt will be using the metrics that I captured to plan current and future projects with.
One half of my project is the node.js side. With that side I have created a node program that connects to a MongoDB database and does any number of Inserts, Updates, Reads and Deletes that I specify. Along with these I have implemented specific timing mechanisms that allow me to gauge performance. In order to gauge performance with large or small transactions, I am able to easily change the code to perform any number of operations.

The other half of my project is the C# side. On this half I have created a program that communicates with a MongoDB and another that communicates with a Microsoft SQL Server. These programs also do Insert, Update, Read and Delete processes with timers in order to gauge performance. My hypothesis for this project was that the node.js to MongoDB would be the fastest setup.
Business Benefit

Once this project is complete and I’ve given my final solution the business will begin development on a program that is slated to save over around $1 million within the next year. Construction Software Technologies operates on a $5 million dollar budget; this project would be offering our company a 20% savings.

![Savings Each Year](image)

Figure 2. Savings Each Year

With this savings we would have more capital available to invest in other areas of our business that show more promise for the future such as predictive analytics to look at trends in construction work and bidding.

Solution

Node.js is a very fast, scalable platform that was built using Google Chrome’s JavaScript runtime. Node.js makes use of an event-driven, non-blocking I/O model that makes it lightweight and efficient. This model can also cause some problems, as I discovered. With node, you get an asynchronous, single threaded experience. This is
great when you have operations that you need done at the same time and don’t really care what order they are done in. Node will start a process, and then move on to the next event that is in line. Node does not inherently care if the next one is the one that should logically, be completed next.

Where I ran into trouble on this was with database transactions that were to be done in succession. With my node.js to Mongo and SQL databases for example, I wanted to insert 5000 items, then update those 5000 rows to something else. Due to the asynchronous nature of node, it wanted to start the insert, then move directly on to the update. So while the 5000 rows were being inserted, they were also be updated, which is not ideal. In order to overcome this problem I was able to create a recursive function. A recursive function is a function that will start and has a step within it to call itself to start again. Within my function, I also have a step that will set a counter to 5000 and with each successful run of the function the counter gets decremented by one. Once the counter is less than one, I call the next function which is update. With this recursive function I was able to solve my problem and control how node flowed through my program. I have the code from my program listed below.

C# is a very mature programming language that offers many ways to tackle different problems. I found my C# apps to be easier to code due to a longer history and more support than JavaScript. C# also offers a much more mature environment with more options on how to handle problems. The benefit of JavaScript is its ease of
programming and the speed at which you can create programs.

```javascript
var sql = require('mssql');
var conn_str = "Driver={SQL Server Native Client 11.0};Server=.;Database=SeniorDesign;Uid=sa;Pwd=Pa$$wOrd;"

sql.query(conn_str, "Truncate table [SeniorDesign].[dbo].[Person]", [], function (err, results) {
  function Insert(counter) {
    if (counter < 1) {
      Update();
      return;
    }
    sql.query(conn_str, "INSERT into Person (FirstName, LastName) VALUES ('Kevin', 'Gilbert')", [], function (err, results) {
       Insert(counter - 1)
    });
  }
  Insert(counter);
})
```  

Figure 3. Node.js to SQL recursive function

C# to SQL and the Node.js to Mongo apps were the easiest to create. Which is what I initially theorized. They are both using the technologies that they were designed for. SQL was designed with C# in mind as MongoDB was originally designed to be used with JavaScript. Examples of my C# to SQL and Node to MongoDB can be found below.

```csharp
using (SqlCommand cmd = new SqlCommand())
{
    cmd.Connection = conn;

    try {
        conn.Open();
        var deleteTimer = new Stopwatch();
        cmd.CommandText = delete;
        cmd.CommandType = CommandType.Text;
        deleteTimer.Start();
        cmd.ExecuteNonQuery();
        deleteTimer.Stop();
        Trace.WriteLine("DELETE TIME:" + deleteTimer.ElapsedMilliseconds);
        conn.Close();
    }
    catch (SqlException e)
    {
        Trace.Write(e);
    }
}
```  

Figure 4. C# to SQL
In my C# code I was able to use a built-in Stopwatch function that C# has readily available. It made the timing of different functions extremely convenient, I also double checked these timings by using SQL and Mongo profiler tools that were built into their respective databases. I found that the profilers were more accurate than the different stopwatches I was able to make use of.
Node to SQL posed its challenges in finding the driver to connect to the SQL database with. SQL was not natively supported by Node and a driver had to be developed for it. Node to SQL offers very decent speeds in everything except Insert, which it was the slowest at. The fastest out of all of the environments was node.js to MongoDB by quite a margin as my charts show below. C# to MongoDB also showed surprisingly good performance, which I did not hypothesize. The delete times on all of the different environments were extremely fast and only took one millisecond or less, on average. To achieve the numbers of my tests I ran each program 50 times, recorded the results and took the averages. On average Node.js to MongoDB was the fastest.

Figure 6. Final Timings
<table>
<thead>
<tr>
<th></th>
<th>Node To SQL</th>
<th>C# to SQL</th>
<th>Node To mongoDB</th>
<th>C# To mongoDb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>3200</td>
<td>1878</td>
<td>200</td>
<td>1117</td>
</tr>
<tr>
<td>Update</td>
<td>34</td>
<td>22</td>
<td>50</td>
<td>63</td>
</tr>
<tr>
<td>Read</td>
<td>190</td>
<td>2309</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Delete</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 6. Final Timings

**Conclusion**

I ran into issues around what my project was to be. My project started out as transitioning a SQL database into a MongoDB and testing performance on that. As I started development into that route, my company saw more of an opportunity with this new project and asked me if I would switch my focus about halfway through my project. I learned a lot from this process on how the scope of a project can change and as a professional I have to handle these unforeseen circumstances with grace. A quote from the Software Architect of my company around this time stuck with me “What? Your project scope changed in the middle of it? That sounds like a great Senior Design
to me.—Software Architect Kevin Upchurch” I believe that this was an important lesson for me to learn and couldn’t think of a better time to have experienced it.

My conclusion and final suggestion were to use node.js with MongoDB. The company has decided to use node.js with RavenDB instead of MongoDB. The reason for this is that RavenDB was built with .Net in mind, which suits Construction Software Technologies due to their majority .Net environment. RavenDB is very similar to MongoDB and offers very similar performance. I would say that my project has been a success and development is already underway on this project which is slated to be fully completed in the summer of 2014.
References

