SQL VERSUS MONGODB FROM AN APPLICATION DEVELOPMENT POINT OF VIEW

by

Ankit Bajpai

B.S, Jawaharlal Nehru Technological University, India, 2010

A REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2015

Approved by:

Major Professor
Doina Caragea

Copyright

Ankit Bajpai

2015

Abstract

There are many formats in which digital information is stored in order to share and re-use it by different applications. The web can hardly be called old and already there is huge research going on to come up with better formats and strategies to share information. Ten years ago formats such as XML, CSV were the primary data interchange formats. And these formats were huge improvements over SGML (Standard Generalized Markup Language). It's no secret that in last few years there has been a huge transformation in the world of data interchange. More lightweight, bandwidth-non-intensive JSON has taken over traditional formats such as XML and CSV.

BigData is the next big thing in computer sciences and JSON has emerged as a key player in BigData database technologies. JSON is the preferred format for web-centric, "NoSQL" databases. These databases are intended to accommodate massive scalability and designed to store data which does not follow any columnar or relational model. Almost all modern programming languages support object oriented concepts, and most of the entity modeling is done in the form of objects. JSON stands for Java Script object notation and as the name suggests this object oriented nature helps modeling entities very naturally. And hence the exchange of data between the application logic and database is seamless.

The aim of this report is to develop two similar applications, one with traditional SQL as the backend, and the other with a JSON supporting MongoDB. I am going to build real life functionalities and test the performance of various queries. I will also discuss other aspects of databases such as building a Full Text Index (FTI) and search optimization. Finally I

will plot graphs to study the trend in execution time of insertion, deletion, joins and co-relational queries with and without indexes for SQL database, and compare them with the execution trend of MongoDB queries.

Table of Contents

Table of Contents v

List of Figures vii

List of Tables x

1 Introduction 1

1.1 Model View Controller (MVC) . . . 2

2 Background 4

2.1 Relational Databases . . . 4

2.1.1 Database Transaction (ACID Properties) . . . 5

2.1.2 Normalization . . . 5

2.2 NoSQL Databases . . . 6

2.2.1 BASE Properties . . . 7

2.2.2 Classification of NoSQL Databases . . . 7

3 Datasets 10

3.1 Yelp Datasets . . . 10

3.2 Normalization of Datasets . . . 12

4 Building the Applications 15

4.1 Setting up Databases . . . 15

4.1.1 Setting up the SQL Database . . . 15

4.1.2 Setting up the MongoDB Database 16

4.2 Loading data 18

4.2.1 Loading Data on SQL Database 18

4.2.2 Loading Data on MongoDB Database 19

4.3 Queries 21

4.3.1 CRUD Queries 21

4.3.2 Join Queries 36

4.4 Full Text Index (FTI) 38

4.4.1 Creating an FTI for a SQL Table 38

4.4.2 Creating an FTI on MongoDB Collection 40

5 Using the Application 43

6 Testing 53

6.1 Web Performance Testing 53

6.2 Load Testing 55

7 Technologies Used 59

8 Lessons Learned and Conclusions 62

Bibliography 64

List of Figures

1.1 MVC Architecture 2

1.2 MVC Architecture: Login View 3

1.3 MVC Architecture: Login Controller 3

4.1 Loading Data on MongoDB 20

4.2 Inserting a row in a SQL table 21

4.3 Inserting a document in a MonogoDB collection 22

4.4 Deleting a row from a SQL table 22

4.5 Deleting a document from a MongoDB collection 22

4.6 Graph to show time taken by an insert query in SQL and MongoDB 23

4.7 Graph to show time taken by a delete query in SQL and MongoDB 24

4.8 Reading a row from a SQL table 25

4.9 Reading a document from a MongoDB collection 25

4.10 Graph to show time taken by a update query in SQL and MongoDB 26

4.11 Update a row in a SQL table 27

4.12 Updating a document in a MongoDB collection 27

4.13 Graph to show time taken by a update query in SQL and MongoDB 28

4.14 SQL query to find MIN/MAX of a field 29

4.15 MongoDB query to find MIN/MAX of a field 29

4.16 Graph to show the time taken by a aggregation query in SQL (Indexed and Non Indexed) and MongoDB 30

4.17 SQL query to find all the row with in a range 31

4.18 MongoDB query to find all the documents with in a range 31
4.19 Graph to show the time taken by a range query in SQL (Indexed and Non Indexed) and MongoDB 32
4.20 Nested SQL query 32
4.21 Nested MongoDB query 32
4.22 Graph to show the time taken by a nested query in SQL (Indexed and Non Indexed) and MongoDB 33
4.23 GROUP BY and HAVING SQL query 34
4.24 GROUP BY and HAVING MongoDB query 34
4.25 Graph to show the time taken by a group by query in SQL (Indexed and Non Indexed) and MongoDB 35
4.26 Join clause in SQL: Finding a business based on location, rating and category in a normalized table 36
4.27 MongoDB Query: Finding a business based on location rating and category . 36
4.28 SQL Query: Finding a business based on location rating and category on a non-normalized table 37
4.29 Searching for a keyword in an FTI created on review JSONs 41
4.30 Finding unique businesses from the output of FTI search 41
4.31 Searching for a keyword in an FTI created on reviews embedded business JSONs 41

5.1 Using the Application: Homepage 44
5.2 Using the Application: Login page 45
5.3 Using the Application: User homepage 46
5.4 Using the Application: Search page 47
5.5 Using the Application: Output of the search 48

5.6 Using the Application: Admin homepage 49
5.7 Using the Application: Business homepage 50
5.8 Using the Application: Create a business page 51
5.9 Using the Application: Edit a business page 52

6.1 Performance testing (user_web): Webpages related to the user role and their average response times 54
6.2 Performance testing (admin_web): Webpages related to the admin role and their average response times 54
6.3 Load test (user_admin): Including web performance tests 55
6.4 Load test (user_admin): Connection speeds included 56
6.5 Load test (user_admin): Browsers included in the load test 57
6.6 Load test (user_admin): The result of the load test 58

List of Tables

4.1 Summarization of Query Performances on SQL and MongoDB 38

7.1 Project Metrics 61

Chapter 1

Introduction

For this report, we are going to build two applications, one with SQL as backend and the other with MongoDB. Both applications are built on MVC (Model View Controller) pattern architecture. The application with SQL as backend will be developed using ASP.Net MVC 4 and the NoSQL application will be built on Java Spring MVC. The reason for choosing two different frameworks is to make sure I am using the most suitable programming language and framework for the respective databases. ASP.Net MVC 4 is very convenient to use for SQL databases as most of the basic controller logic and front-end pages are created once the database model is created. Similarly, Java Spring has proven to be very convenient to use with MongoDB, as the helper library for MongoDB in Java is well tested and optimized.

In this application we have two types of user roles “User” and “Admin”. The admin can create, edit and delete a user or a business (e.g., Chipotle, J’s saloon, etc.) from the database. The User can search and write reviews on a business. The User can also search for a business based on different criteria such as location, rating and categories (e.g., bar, indian, haircut, etc.) of the business, or search for a keyword on a full text index generated on the reviews.

1.1 Model View Controller (MVC)

As stated before, both applications are built on the MVC pattern architecture. MVC stands for Model-View-Controller. Here the Model represents the application core (for instance a list of database records). The View displays the data (the database records). The Controller handles the business logic. Below is a brief description of how the MVC pattern works. To explain the working of MVC we take an example of login functionality[1].

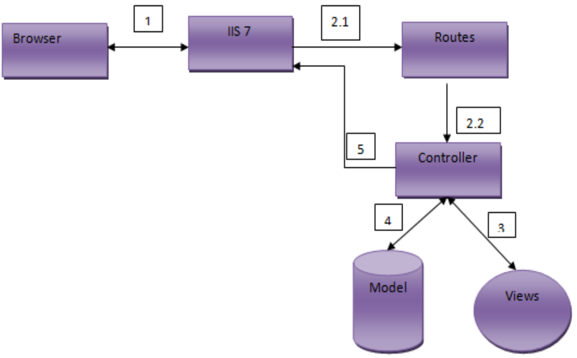

Figure 1.1: *MVC Architecture*

- In Figure 1.1 Step 1, the browser communicates with the IIS (Internet Information Services) server by means of the HTTP protocol which runs on TCP/IP. All the information is passed in the form of URLs. For example, in the current application, when a user is attempting to login, the user first enters the credentials in a view (Figure 1.2). A post request is generated by the browser and is sent to IIS with the help of the HTTP protocol.

- In Figure 1.1, Steps 2.1 and 2.2, the IIS will contact the routes and find out the place (controller) to process this request. For the above example the request is processed at AccountController. Figure 1.3 shows the code of the function which will be called in AccountController (You can see that this function handles POST request).

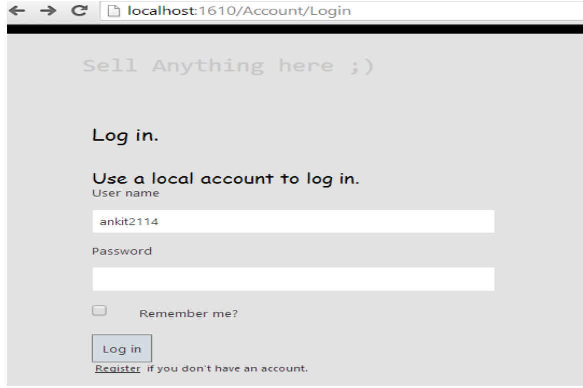

Figure 1.2: *MVC Architecture: Login View*

```
//
// POST: /Account/Login

[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
0 references
public ActionResult Login(LoginModel model, string returnUrl)
{
    if (ModelState.IsValid && WebSecurity.Login(model.UserName, model.Password, persistCookie: model.RememberMe))
    {
        return RedirectToLocal(returnUrl);
    }

    // If we got this far, something failed, redisplay form
    ModelState.AddModelError("", "The user name or password provided is incorrect.");
    return View(model);
}
```

Figure 1.3: *MVC Architecture: Login Controller*

- In Figure 1.1, Step 3 you can see that the controller accesses the login database and validates the input data.
- In Figure 1.1, Steps 4 and 5, based on the sanity of the username and the password, the appropriate view is sent to the IIS server, which in turn sends it back to the browser for user to view it.

Chapter 2

Background

2.1 Relational Databases

A database is defined as a collection of persistent data that is used by the application systems of a given enterprise. More often than not, the term Database has been misused to refer the Database Management Systems (DBMS). DBMS is a systems which provides user with features such as programming interface and transaction management. It was not until 1960s when the DBMS was available for businesses to store their data electronically. Before that data was stored in books or punch cards[2].

These days two most widely used types of database are relational databases and NoSQL databases. Although NoSQL is relatively new (introduced in year 2000) compared to other databases, it has gained popularity because of its ability to handle unrelated and unstructured data. Relational databases need a fixed schema to store data, as opposed to NoSQL which does not impose any such constraints. Although the two types of databases differ in many aspects, with little change in the implementation of an application both these databases can be used to perform similar functionalities. In this report I will contrast and compare the two types of databases. I will be using Microsoft SQLlite as a relational

database and MongoDB as a NoSQL database.

2.1.1 Database Transaction (ACID Properties)

An important aspect of a relational database is to guarantee that database transactions are processed reliably. This requirement led to ACID properties. Below is a short description of these properties[3].

- **Atomicity**: A transaction is made up of one or more tasks. The atomicity property makes sure that either all the tasks in the transactions get executed or non of them.
- **Consistency**: The consistency property ensures that the database remains in a consistent state before the start of the transaction and after the transaction is over (whether successful or not).
- **Isolation**: This property makes sure that no task gets access to the data which is already involved in other transaction. This is carried out with the help of locks, if a data is being used in update or write operation of a transaction then that transaction holds a lock on that data. Only the transaction which holds the lock can access the data.
- **Durability**: Durability refers to the property which guaranties that database remains in a stable state before and after a system failure. Usually this is achieved by maintaining logs.

2.1.2 Normalization

An important design aspect of relational databases is the normalization of the schema. Normalization process eliminates redundancy and makes sure that the data remains consistent. Any database needs to be in BCNF (Boyce Codd normal form) in order to avoid duplication

and maintain consistent data[2]. But sometimes its not possible to normalize database to BCNF, in that situation we can settle for lower normalized form such as third normal form (3NF). To full understand the BCNF we first need to look at the 3NF. A relation is in 3NF if the following conditions satisfy.

- All the fields contain atomic value only.
- Every non key attribute should be dependent on the super key.

BCNF is slightly stronger rule than the 3NF, for a relationship to be BCNF every attribute of the relationship should be dependent on the super key.

2.2 NoSQL Databases

NoSQL stands for "Not Only SQL". The NoSQL databases do not impose any constraints on storing data (schema-less). This also means that unlike relational databases the data need not be normalized. NoSQL databases have become more popular in recent years due to the introduction of Web 2.0[4]. The applications these days need to handle a lot of unstructured data, this includes images and videos. Since such web applications are very agile, underlying databases have to be flexible as well in order to support the unstructured data. Adding or removing a feature to a blog is not possible without system un-availability if a relational database is being used. But NoSQL databases can enable such user concurrency.

Due to the flexible nature of NoSQL it is very difficult to ensure stringent ACID properties over the data stored in NoSQL databases. So engineers for NoSQL came up with BASE properties. BASE stands for BasicallyAvailable, Soft state and Eventual consistency. This was based on CAP theorem, CAP stands for Consistency, Availability and Partition tolerance. This theorem states that a distributed system can hold only two of the following three properties[5].

- Consistency : All the nodes in the distributed system see the same data.
- Availability : All the nodes send a response to every request indicating success or failure.
- Partition Tolerance : Even if one or more nodes fail within the distributed system, the rest of the system should function properly.

2.2.1 BASE Properties

- **Basically Available**: This property corresponds to the availability property of the CAP theorem. If there is a request to access certain data, irrespective of the availability of the data a response has to be sent to the requesting node. The response can be either "success" or "failure" [5].
- **Soft state**: As NoSQL databases have distributed architecture, there is propagation of updates to a node even when there is no explicit input from a user to that node. This propagation of updates correspond to the "Eventual Consistency" property of NoSQL databases. So data in particular node needs to be regularly refreshed in order to reflect all the changes. This property is referred as "Soft State" [5].
- **Eventual consistency**: In NoSQL databases when there is an update in data, the update might not be propagated to all the nodes immediately. But eventually these updates are reflected on all the nodes. This property of NoSQL databases is referred as "Eventual Consistency" [5].

2.2.2 Classification of NoSQL Databases

NoSQL databases can be broken down into four major categories and in this section I will briefly discuss them.

- **Key-Value Stores:** The Key-Value type of NoSql databases has a very simple data model. As the name suggests the data is stored in the form of key-value pairs. The information is searched against a key with the help of basic get functions which take “key” as argument and returns the corresponding value. For example, if I have a key-value entry in my NoSQL database $<$ “*name*″, “*Ankit*″ $>$ then get (“name”) would return “Ankit”. This data model is wrapped with mechanism to accommodate BASE properties[6].

- **Document Stores**: The Document based NoSQL databases are very interesting because instead of rows and columns your data is stored in documents. These documents are simply stored in the form of JSON files. Here each record is treated as a document, and all the documents are independent to one another. That means that one document might have entirely different set of fields when compared to any other document stored in database. Document based NoSQL databases such as MongoDB have very powerful query engines and indexing features that make it easy and fast to execute many different optimized queries. Also, the application logic is easier to write as queries return the results in the form of JSON and these can be easily converted to objects. This way we don‘t have to translate between objects in our application and queries[8].

- **Wide Column Stores**: These types of NoSql databases make use of a hybrid approach by mixing the declarative characteristics of a relational database with key-value pairs of NoSql databases. Wide Column database store data as sections of columns of data rather than as rows of data. The way this hybrid works is, we need to declare a group of column as part of reliable schema. And this column group in turn can have different columns for each row.

- **Graph Databases**: This kind of NoSQL databases are the most specialized kinds of

databases. They are focused on the relationship between entities rather than entities themselves. These types of databases use nodes, edges and properties to represent data. The nodes represent entities such as business or users. Properties are attributes that are related to nodes. For example, if you have a business node then name of the business, type of business are the properties of that node. Edges are the lines which connects one node to another. Meaningful patterns emerge when we examine the connection of nodes, properties and edges.

Chapter 3

Datasets

3.1 Yelp Datasets

The datasets used in this project were provided by "Yelp.com". These datasets are available for any academic project and can be downloaded from this link, https://www.yelp.com/academic_dataset. These datasets are in the form of three JSON files. The first is the business JSON. Business objects contain basic information about local businesses. There are 13690 different businesses listed in business dataset. The fields are as follows:

```
{         business_id: (business_id does not identify
                              all the business uniquely),
           name: (the full business name),
          full_address: (address),
          city: (city),
          state: (state),
          latitude: (latitude),
          longitude: (longitude),
```

```
        stars: (average rating given by the user),
        review_count: (review count),
        photo_url: (photo url),
        categories: (category names),
        url: (yelp url)
}
```

The review JSON contains information about the reviews such as the review text and the rating. The user_id can be used to associate reviews on different businesses by the same user. There are 330071 reviews in the review dataset.

```
{       business_id: (business_id),
        location_id: ( Identifier for business location
                                [longitude, latitude]),
        review_id:(the identifier for review)
        user_id: (the identifier of the authoring user),
        stars: ( star rating, integer 1-5),
        text: (review text),
        date: (date, formatted like 2011-04-19),
}
```

Lastly, we have the customer JSON. There are 56640 customers in this JSON file.

```
{       user_id: (unique user identifier),
        name: (first name, last initial, like Matt J.),
        address:(customer address),
        state:(customer state),
        review_count: (review count),
        average_stars: (floating point average, like 4.31)
```

```
}
```

3.2 Normalization of Datasets

I will try to normalize earlier discussed datasets to BCNF. Let us consider the business JSON, this holds information about the businesses. But an important point to notice here is that the business_id is not the primary key. You can have a business franchises with same business_id but different locations. For example, "Chipotle Mexican Grill" business has the same business_id for all of its restaurants at different locations. Here the "name", "categories", "photo_url","url" depends on the "business_id", but rest of the fields are dependent on the location. So we will split this JSON into two tables, one with all the attributes which are dependent on "business_id" and the "business_id" (primary key for this table). And the other table with the rest of the fields. Also, the categories field in the business dataset is an array. So in order to make the field values atomic I have concatenated all the values to form a single string. We will also include a new field "location_id" in the second table. The second table will have the combination of "location_id" and "business_id" as the primary key. Even though the current table structure is not in BCNF (stars and review_count fields does not depend on the primary key business_id and location_id), I have not further normalized the database. This reason to stop the normalization was to make sure that I do not increase the number of tables and in turn increase the query execution time (Join clause cause huge decrease in query performance). And, as the insertion and updation of a business will happen far lesser time then searching a business in this application, this decision seems reasonable. Below are the model classes after normalizing the business JSON.

```
public class Business
    {
        public string business_id   \\ Primary key
```

```
    public string categories
    public string photo_url
    public string name
    public string url
}
public class Location
{
    public string business_id  \\Primary key
    public string location_id \\Primary key
    public string full_address
    public string city
    public double longitude
    public double latitude
    public string state
    public double stars
    public int review_count

}
```

Review JSON contains non atomic values for "location_id" field. This is a combination of latitude and longitude. In order to normalize, we will replace this with the "location_id" from the location table. The "User" JSON is already in 3NF. Below are the normalized model classes for review and user JSONs.

```
 public class Review_text
{
    public string review_id  \\ Primary Key
    public string business_id
    public string location_id
    public string user_id
    public int stars
    public string text
    public string date
    public string type
}

 public class User
{
    public string user_id  \\ Primary Key
    public string name
    public string address
    public string state
    public string review_count
    public string average_stars
}
```

Chapter 4

Building the Applications

The first step for building the application was to set up the database servers. The .Net MVC 4 application comes with Microsoft SQL compact installed. Also, the connection string to SQL is set up by the MVC 4.0 framework by default. In the case of MongoDB the database had to be installed and connection string to be made. In this chapter, I will compare installation, querying and efficiency of both the databases. The efficiency will be measured in terms of the query execution time.

4.1 Setting up Databases

4.1.1 Setting up the SQL Database

As discussed before, I have used an inbuilt Microsoft SQL compact as my relational database. The connection is made to the database through a connection string which is defined in the web.config file of .Net MVC 4.0 application folder. Below is the connection string used in this application.

```
<connectionStrings>
<add name="DefaultConnection"
```

```
connectionString="Data Source= (LocalDb)\v11.0;
Initial Catalog=aspnet-Project\_20140413003109;Integrated
Security=SSPI;
AttachDBFilename=|DataDirectory|
\aspnet-Project_20140413003109.mdf"
providerName="System.Data.SQLClient" />
</connectionStrings>
```

The connectionString tag defines critical properties for the connection to be established, and defines the location for the .mdf file. The providerName holds the name of the database to connect to. You can have multiple .mdf files on multiple servers, in which case multiple connectionString tags should be defined.

4.1.2 Setting up the MongoDB Database

For setting up the MongoDB datadase, I needed to install the MongoDB database and then set up a connection string to make sure the application can access the database. I used Homebrew (a software which downloads and installs the latest version of all the open source softwares) to install MongoDB and set up the database. Below are the steps I followed:

- Install MongoDB database using the following command.

  ```
  Ankits-MacBook-Pro:~ ankitbajpai$brew install MongoDB
  ```

- Once MongoDB was installed, a database and collections were created so that data can be loaded. In order to create a database we need to logon to MongoDB prompt first. The command to do that is "mongo".

  ```
  Ankits-MacBook-Pro:~ ankitbajpai$ mongo
  MongoDB shell version: 2.6.4
  ```

```
Server has startup warnings:
2014-10-04T17:32:05.269-0500 [initandlisten]
# Creates a new database and makes it ready to use
>use yelp
switched to db yelp
>
```

- Once the server was set up, now the next step is to create a connection string. I used the helper libraries listed below in order to connect the Java Spring application to the database and perform operations.

```
#Helps with database level operations
import com.MongoDB.DB;
# Helps with collection level operations
import com.MongoDB.DBCollection;
#Helps in handling query result
import com.MongoDB.DBCursor;
# Helps establishing connection to the database server
import com.MongoDB.MongoClient;
# Connects to MongoDB
MongoClient mClient = new MongoClient ("localhost", 27017);
#Sets the database to yelp.
DB db=mClient.getDB ("yelp");
```

4.2 Loading data

In this section, we will discuss the process to load the datasets into SQL and MongoDB databases.

4.2.1 Loading Data on SQL Database

As our data is in the form of JSON, we needed to parse the JSON and load the data one tuple at a time. Most of the time when we need to load data in the SQL database, we need to parse and clean the data, even if the dataset is in the form of XML or plain text. Following were the steps and time taken in order to load the data.

- Create a temporary class which can hold the data from the JSON. This class will only be used during parsing and loading the data.
- Now, we need to read one JSON at a time, parse and load it into the database.

```
# Reads and copies all JSON from input files
StreamReader r = newStreamReader (”../Desktop/business_data.txt”);
// Reading One JSON at a time
while ((JSONline = r.ReadLine())!= null){
#Parsing JSON into business objects
business bus = JSONConvert.DeserializeObject<business>(JSONline);
#Loading Data
context.restaurant.AddOrUpdate(x =>x.business_id ,new Restaurant )]
{
            business_id = bus.business_id ,
            full_address = bus.full_address
}
```

It took about 22 minutes to perform the above steps. Following is the output from the program which parsed and loaded the JSON file onto the SQL database.

```
#####   Seed method has started    11:26:49 PM     #######
#####   Total number of tuples inserted = 13483    #######
#####   Seed method has ended    11:48:51 PM     #######
```

4.2.2 Loading Data on MongoDB Database

MongoDB is a document database, it stores data in the form of JSON documents. As discussed before, JSON provides a rich data model that seamlessly maps to native programming language types, and the dynamic schema makes it easier to evolve our data model as compared to a system with enforced schemas such as RDBMS. Thus, in order to load the dataset into MongoDB we need not parse anything. Following are the commands used to load into MongoDB

- Start the MongoDB Server using the"Mongod" command.
- MongoImport is the utility which is used to load data into MongoDB.

```
Ankits-MacBook-Pro:~ ankitbajpai$
 # Selects the database
mongoimport --db yelp
#Creates a collection to load data
 --collection business
 #Specify the location of data
--file /Users/ankitbajpai/Desktop/business.JSON
--JSONArray
```

```
insert yelp.business ninserted:1 keyUpdates:0 numYields:0 locks(micros) w:40 305ms
insert yelp.business ninserted:1 keyUpdates:0 numYields:0 locks(micros) w:120922 120ms
insert yelp.business ninserted:1 keyUpdates:0 numYields:0 locks(micros) w:139635 139ms
```

Figure 4.1: *Loading Data on MongoDB*

In Figure 4.1 you can see that the import took place in 3 operations and the summation of these operations comes to 564ms (305ms+120ms+139ms). As expected MongoDB was way faster when compared to SQL interms of loading data, as MongoDB directly stores each JSON as a document, while in SQL we needed to parse each JSON and load the data one row at a time.

4.3 Queries

In this section, we will discuss various queries which were written in the development of this application. The queries can be mainly categorized as follows: CRUD (Create, Read, Update and Delete) queries and Join queries. We will also compare the execution time for each of these queries in MongoDB and SQL. We should note that each query which is presented is executed 100 times and an average is taken to get an accurate execution time. I also made sure that I change the query selection criteria for every execution to get the most optimal result. For example, if I am executing a ready query, then ever time I run the query I would read a different row. In the same way for creating trend graph I have run the queries on SQL and MongoDB databases for 100 times, 500 times, 1000 times, 5000 times and 10000 times, and recorded respective execution time.

4.3.1 CRUD Queries

CRUD queries are used in various places and they form the backbone of this application. For example, an Admin can add a business (create query), delete a business (delete query), update business information (update) or retrieve information about a business (read query).

Figure 4.2 shows a query to insert a new business into the business table and its average execution time on SQL server. Here "business" is the table name and business_id, categories etc. are the fields. Figure 4.3 shows a query to insert a new business in MongoDB database and its respective average execution time. The average execution time for insert query in SQL is 23.1 ms (Milliseconds), and in MongoDB it is 1.6 ms.

```
Execution time  Query
23.1ms          insert [dbo].[business]{[business_id], [categories], [photo_url], [name], [url]}
                        values {@0,@1,@2,@3}
```

Figure 4.2: *Inserting a row in a SQL table*

Insert: [business] 1,687 µs (33 %) 1,687 µs
33.7% - 1,687 µs - 1 hot spot inv. com.mongodb.DBCollection.insert
33.7% - 1,687 µs - 1 hot spot inv. Mongo_restaurant.Mongo_restaurant.controller.searchController.business_added
33.7% - 1,687 µs - 1 hot spot inv. URL: /mongo_restaurant/admin/add_business

Figure 4.3: *Inserting a document in a MonogoDB collection*

Figure 4.4 shows a query to delete an existing business from the business table and its average execution time on SQL server, while Figure 4.5 shows a query to delete an existing business from the business collection and its average execution time in MongoDB database. The SQL query takes 14 ms to execute, whereas the MongoDB query takes .39 ms.

```
Execution time   Query
14ms             delete[dbo].[business]
                 where ([business_id = @0])
```

Figure 4.4: *Deleting a row from a SQL table*

Delete: [business] {} 390 µs (0 %)
0.1% - 390 µs - 1 hot spot inv. com.mongodb.DBCollection.remove
0.1% - 390 µs - 1 hot spot inv. Mongo_restaurant.Mongo_restaurant.controller.searchController.del_business_p
0.1% - 390 µs - 1 hot spot inv. URL: /mongo_restaurant/admin/del_business

Figure 4.5: *Deleting a document from a MongoDB collection*

Figure 4.6 represents a graph which shows performance of SQL insert query against MongoDB insert query. While Figure 4.7 represents a graph which shows performance of SQL delete query against MongoDB delete query. You can see that as the number of records deleted or inserted increases the execution time increases exponentially for the SQL queries. One reason for such a huge difference in the execution time between the SQL and MongoDB queries is the indexes created on the SQL table. Earlier in this report we created an index to enhance the performance of a query, but creating indexes on a non unique column (In this

case review_count and average_stars) decreases the performance of the inserts and deletions. Now we will look at some aggregation queries, and there execution trends.

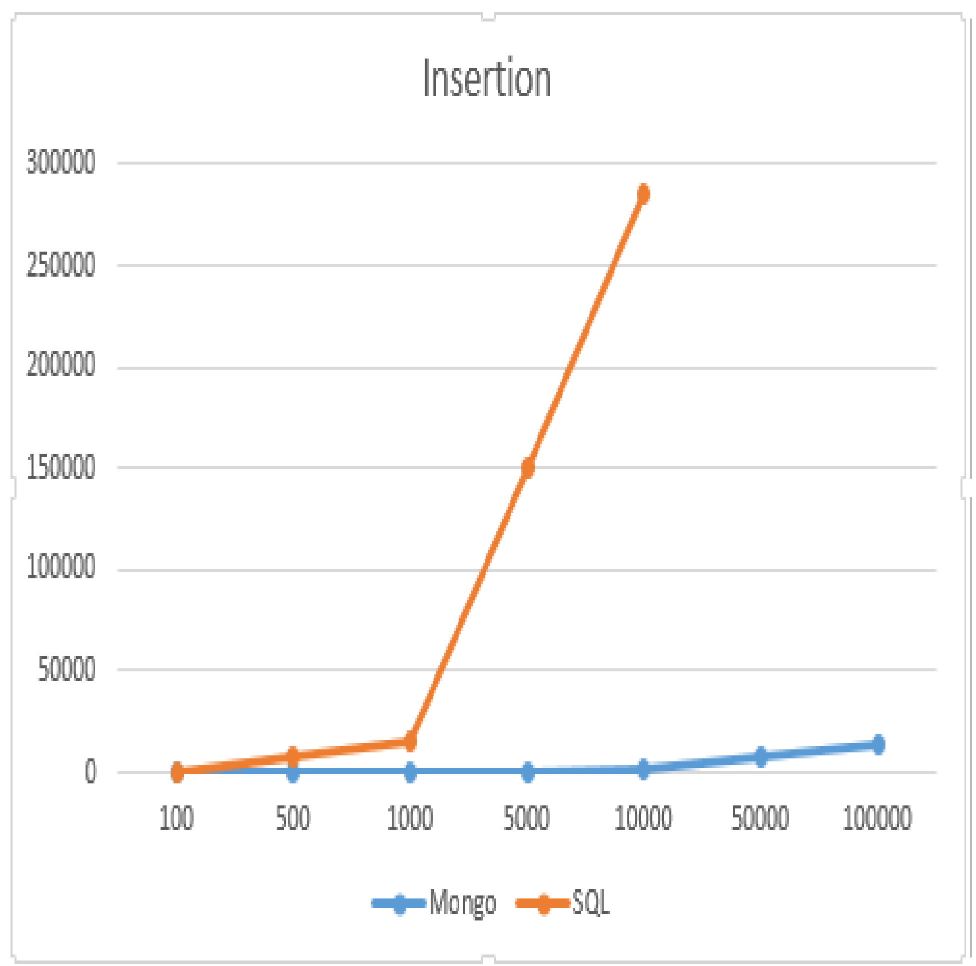

Figure 4.6: *Graph to show time taken by an insert query in SQL and MongoDB (X-axis represents number of times query was executed and Y-axis represents time in milliseconds)*

Figure 4.8 shows a simple query to read a business based on business_id and its execution time on SQL server, while Figure 4.9 shows a query to read a query on MonogoDB server. You can see that the SQL query takes 1.26 ms to execute, whereas the MongoDB query takes 6.7 ms. Figure 4.10 represents a graph which shows performance of SQL read query against MongoDB read query. You can see that reading a row in SQL is faster than reading a row in MongoDB database. As the above SQL read query uses primary key to find the

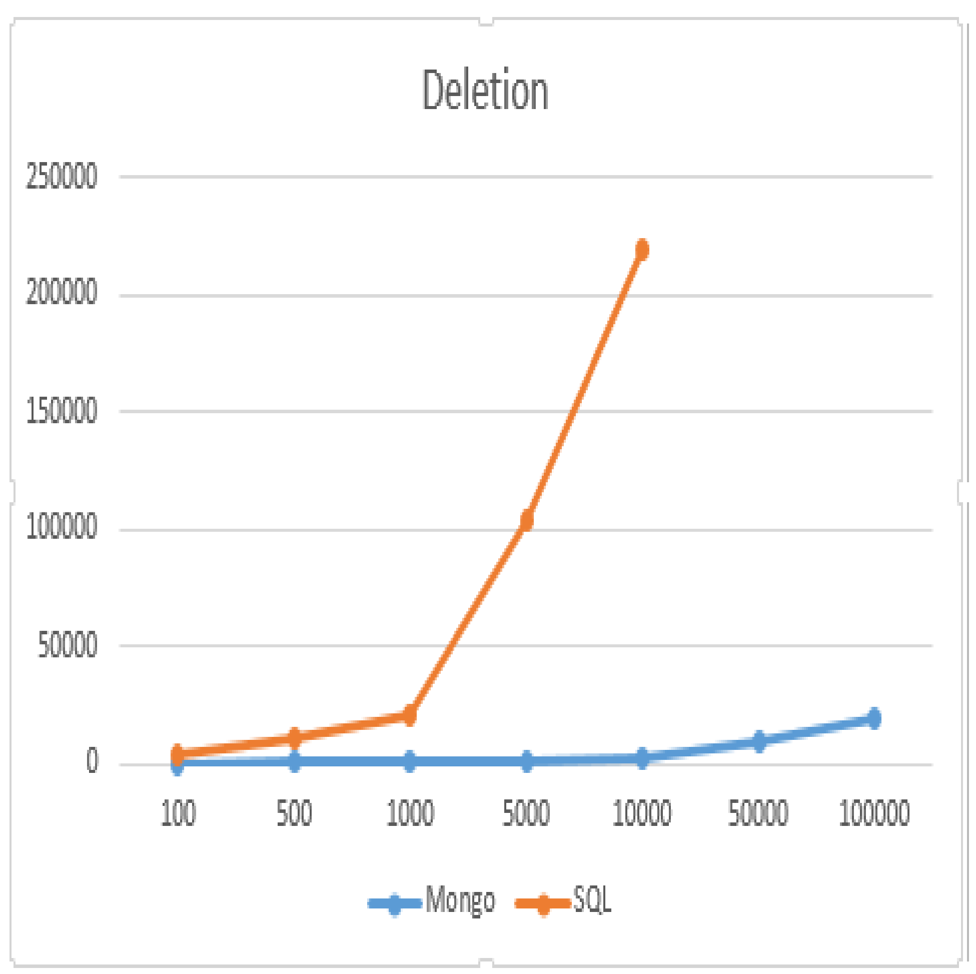

Figure 4.7: *Graph to show time taken by a delete query in SQL and MongoDB (X-axis represents number of times query was executed and Y-axis represents time in milliseconds)*

row to read, it is faster than MongoDB read query.

```
Execution time  Query
1.26 ms         SELECT
                [Extent1].[business_id] AS [business_id],
                [Extent1].[categories] AS [categories],
                [Extent1].[photo_url] AS [photo_url],
                [Extent1].[name] AS [name],
                [Extent1].[url] AS [url],
                FROM [dbo].[business] AS [Extent1]
                WHERE [Extent1].business_id =@0
```

Figure 4.8: *Reading a row from a SQL table*

```
Query: [business] find {business_id: ?}                                    6,755 µs (1 %)
  1.3% - 6,755 µs - 1 hot spot inv. com.mongodb.DBCollection.findOne
    1.3% - 6,755 µs - 1 hot spot inv. Mongo_restaurant.Mongo_restaurant.controller.searchController.review
      1.3% - 6,755 µs - 1 hot spot inv. URL: /mongo_restaurant/user/review
```

Figure 4.9: *Reading a document from a MongoDB collection*

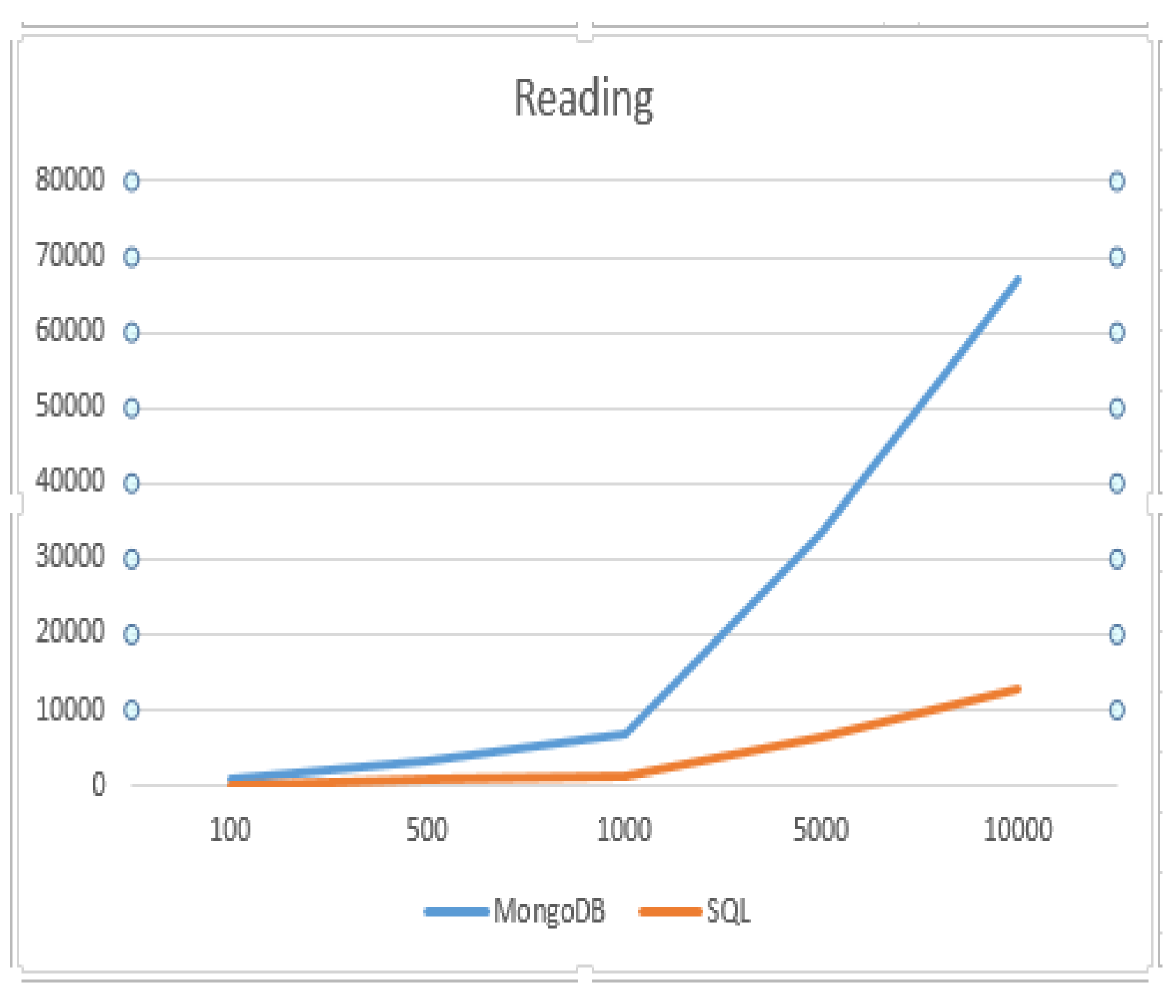

Figure 4.10: *Graph to show time taken by a read query in SQL and MongoDB (X-axis represents number of times query was executed and Y-axis represents time in milliseconds)*

Figure 4.11 shows an SQL query to update review count busing business_id and location_id, and its execution time, while Figure 4.12 shows similar MongoDB update query and its average executing time. Here SQL query takes 1.35 ms where as MongoDB update query takes only 2.492 ms.

Execution time	Query
1.35 ms	Update [dbo].[Location] set review_count = @0 WHERE business_id = @1 and location_id = @2

Figure 4.11: *Update a row in a SQL table*

```
Update: [business] {_id: ?, business_id: ?, full_address: ?, ... 2,492 µs (1 %)
  1.9% - 2,492 µs - 1 hot spot inv. com.mongodb.DBCollection.update
    1.9% - 2,492 µs - 1 hot spot inv. Mongo_restaurant.Mongo_restaurant.controller.searchController.Review_add
      1.9% - 2,492 µs - 1 hot spot inv. URL: /mongo_restaurant/user/review
```

Figure 4.12: *Updating a document in a MongoDB collection*

Figure 4.13 represents a graph which shows performance of SQL update query against MongoDB update query. You can see that updating a row in SQL is faster than updating a row in MongoDB database. As the above SQL update query uses primary key to find the row to be updated, it is faster than MongoDB update query.

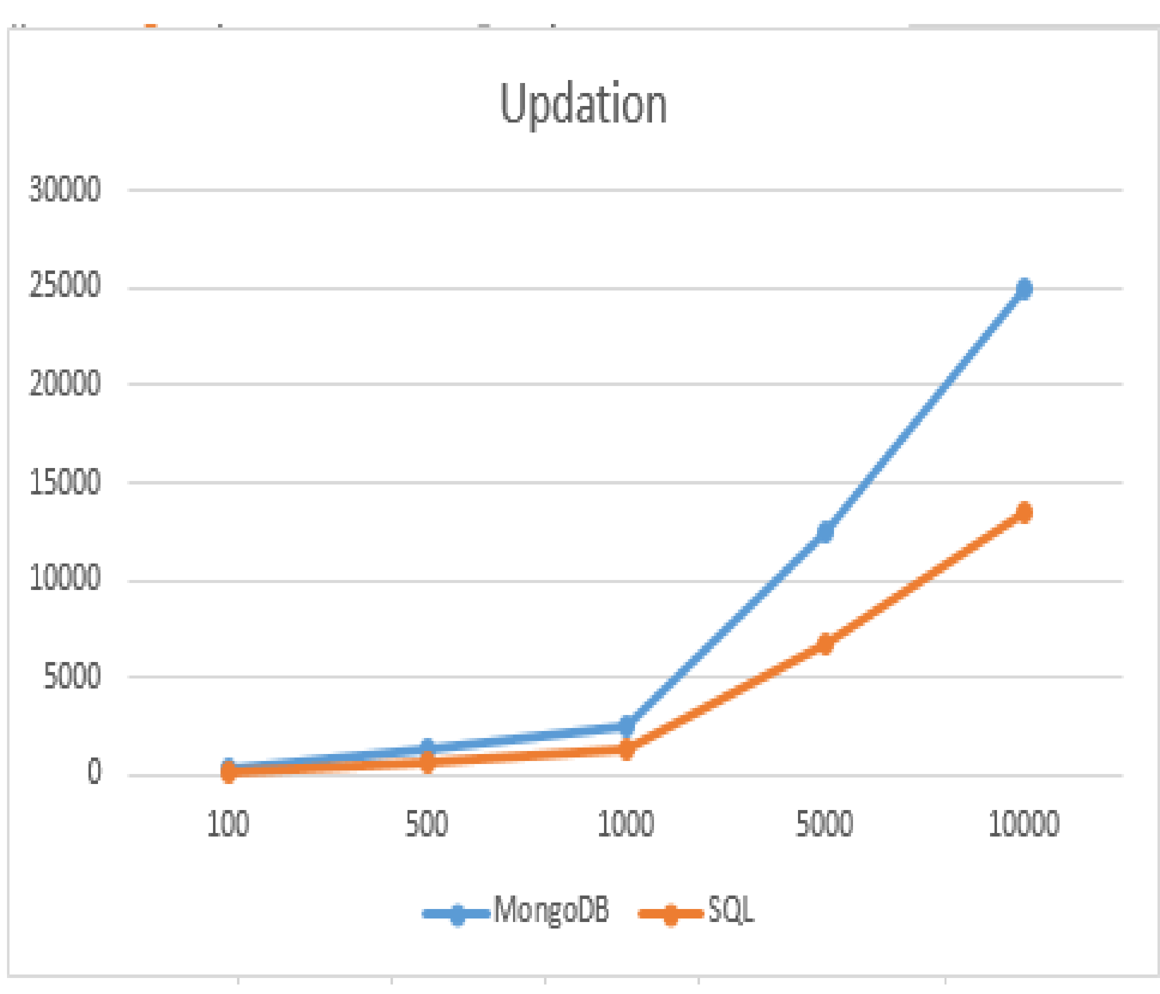

Figure 4.13: *Graph to show time taken by a update query in SQL and MongoDB (X-axis represents number of times query was executed and Y-axis represents time in milliseconds)*

Finding maximum/minimum review_count from the user table. For SQL query we are going to use the aggregation function "MAX/MIN", but in MongoDB we do not have any such aggregation functions. We need to sort all the documents based on review_count field and then return the first/last document. Figure 4.14 shows an SQL aggregation query and its execution time. While Figure 4.15 shows similar MongoDB aggregation query and its average executing time. Here SQL query takes 17.06 ms where as MongoDB update query takes only 36.76 ms.Figure 4.16 represents a graph which shows the performance of SQL aggregation query against similar MongoDB query. You can see that SQL performs better than MongoDB query as review_count is indexed.

```
Execution time  Query
17.03           Select MAX([Extent1].[review_count])
                FROM business as Extent1
```

Figure 4.14: *SQL query to find MIN/MAX of a field*

```
Execution time  Query
36.78           [business] {{$sort:review_count}, Max:{1}}
```

Figure 4.15: *MongoDB query to find MIN/MAX of a field*

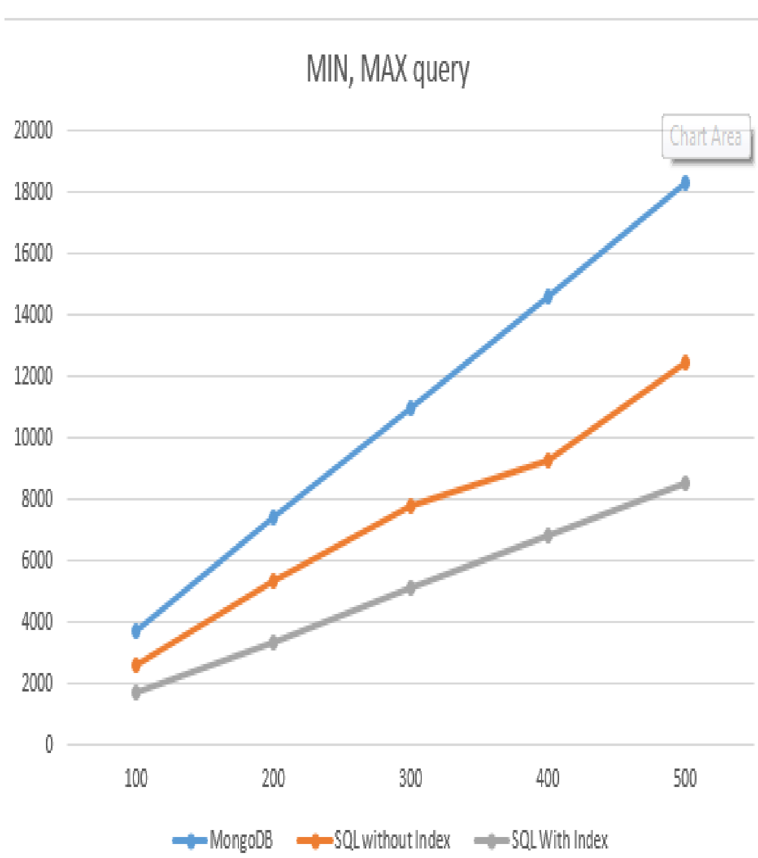

Figure 4.16: *Graph to show the time taken by a aggregation query in SQL (Indexed and Non Indexed) and MongoDB (X-axis represents number of times query was executed and Y-axis represents time in milliseconds)*

Finding all the users with average_stars between 3 and 4. Figure 4.17 shows an SQL range query and its execution time. While Figure 4.18 shows similar MongoDB range query and its average executing time.Figure 4.19 represents a graph which shows the performance of SQL range query against similar MongoDB query. You can see that SQL performs better than MongoDB query as average_stars is indexed.

Execution time	Query
30.36	Select [Extent1].[name] Where [Extent1].[average_stars] BETWEEN @1 AND @2 FROM [dbo].[users]

Figure 4.17: *SQL query to find all the row with in a range*

Execution time	Query
36.78	[business] {user:$name, average_stars:{$gt:3, $lt:4}}

Figure 4.18: *MongoDB query to find all the documents with in a range*

Finding all the user_id with there average_stars greater than or equal to the average of average_stars of all the users (Nested query). Figure 4.20 shows an SQL nested query and its execution time. While Figure 4.21 shows similar MongoDB nested query and its average executing time.Figure 4.22 represents a graph which shows the performance of SQL range query against similar MongoDB query. Here MongoDB query is faster than that of SQL query because for calculating the average of a field, entire table needs to be read. Creating an index would not enhance the performance of SQL query in this case.

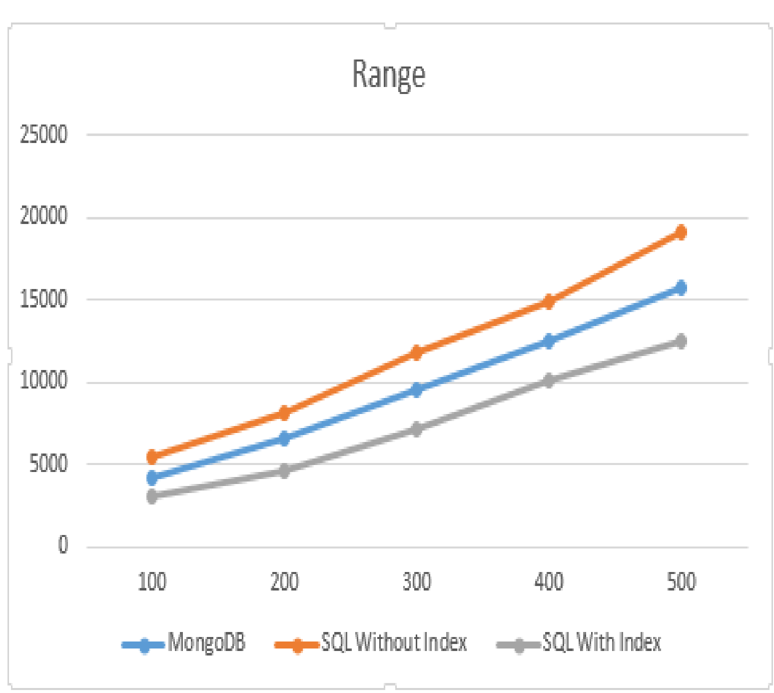

Figure 4.19: *Graph to show the time taken by a range query in SQL (Indexed and Non Indexed) and MongoDB (X-axis represents number of times query was executed and Y-axis represents time in milliseconds)*

```
Execution time  Query
22.11           Select [Extent1].[user_id]
                Where [Extent1].[average_stars]
                > { Select AVG([Extent2].[average_stars]
                    From [dbo].[users] As Extent2
                From [dbo].[users] As Extent1
```

Figure 4.20: *Nested SQL query*

```
Execution time  Query
17.61           [business] {user:$user_id, average_stars:{$gt:{$avg:$average_stars}}
```

Figure 4.21: *Nested MongoDB query*

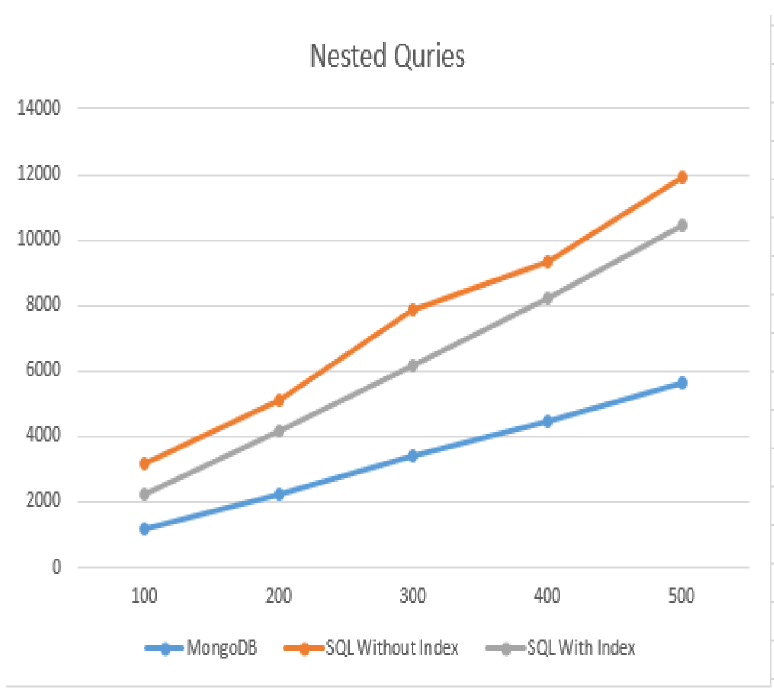

Figure 4.22: *Graph to show the time taken by a nested query in SQL (Indexed and Non Indexed) and MongoDB (X-axis represents number of times query was executed and Y-axis represents time in milliseconds)*

Finding all the states where the sum of the review_count for the business is more than 10000. Figure 4.23 shows an SQL *group by* query and its execution time. While Figure 4.24 shows similar MongoDB *group by* query and its average executing time. Figure 4.25 represents a graph which shows the performance of SQL range query against similar MongoDB query. In this case as well MongoDB query is faster than that of SQL query because *having* and *group by* also require entire table to be read. As in the case of nested query, indexing a column would not enhance the SQL query in this case as well.

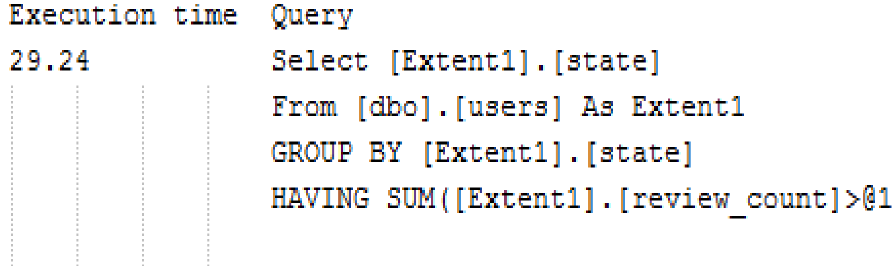

Figure 4.23: *GROUP BY and HAVING SQL query*

```
Execution time  Query
19.97                [business] {{$group:$state}, average_stars:{$match:{$sum{$gt:10000}}}}
```

Figure 4.24: *GROUP BY and HAVING MongoDB query*

Figure 4.25: *Graph to show the time taken by a group by query in SQL (Indexed and Non Indexed) and MongoDB (X-axis represents number of times query was executed and Y-axis represents time in milliseconds)*

4.3.2 Join Queries

SQL join clause is used to combine data from two or more tables in a relational database. Joins combine rows from different tables with the help of a common field. In this application user can search for a business based on minimum rating, state and the category of business. We need to gather information from two different tables in order to come up with results. In this application the “Restaurant” table holds the details about a business, “Location” table holds information about the location and rating of a business. As in MongoDB we do not have joins, all the information related to a business such as location and rating is stored in a single document.

```
Execution time  Query
648 ms          from a in db.business
                join l in db.location on a.business_id equals l.business_id
                where (a.categories.Contains(Query) &&
                l.stars > min &&
                l.state.Equals(state))
                orderby l.state
                ( SELECT a.name, a.url, a.photo_url,
                         l.location_id, l.stars, l.longitude, l.latitude)
```

Figure 4.26: *Join clause in SQL: Finding a business based on location, rating and category in a normalized table*

```
▼ Query: [business] count {stars: {$gt: ?}, categories: {$regex: ?}, state: ?}   60,018 µs (57 %)
  ▼ 57.2% - 60,018 µs - 2 hot spot inv. com.mongodb.DBCursor.count
    ▼ 57.2% - 60,018 µs - 2 hot spot inv. Mongo_restaurant.Mongo_restaurant.controller.searchController.sendoutput
        57.2% - 60,018 µs - 2 hot spot inv. URL: /mongo_restaurant/search
```

Figure 4.27: *MongoDB Query: Finding a business based on location rating and category*

Figure 4.26 shows a SQL query which uses join clause, here data is collected from two different tables. The average execution time for this query was 648.3 ms. It is important to keep in mind that the SQL queries perform better when the fields in where clause are indexed. Without creating index on the *stars* and *state* fields of the location table, the same query took 2861 ms for execution. In Figure 4.27 you can see the MongoDB query to perform the same task. As mentioned earlier MongoDB does not support joins, hence you can see that search is on a single document collection "business". The average execution time for MongoDB query is 60 ms.

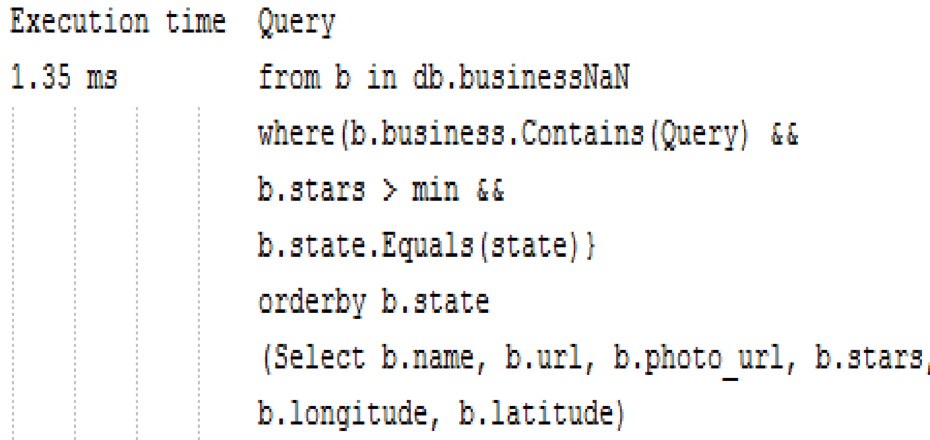

Figure 4.28: *SQL Query: Finding a business based on location rating and category on a non-normalized table*

Figure 4.28 shows a SQL query to find business based on category and rating. Important thing to notice here that this query is run on a non-normalized table. Hence, there is no joins which brings down the execution time. You can see that the average execution time of the query shown in Figure 4.28 is 48.14 ms.

Table 4.1 shows a summary of all the queries run on SQL and MongoDB for this report. It records execution time for SQL query with index on field other than primary key, SQL query with index only on primary key and execution time on MongoDB.

Query Type	Execution Time on SQL (with indexing on fields other than primary key)	Execution Time on SQL (Indexing on primary key only)	Execution Time on MongoDB
Insert Query	N\A	23.1 ms	1.68 ms
Delete Query	N\A	14 ms	.39 ms
Read Query	N\A	1.26 ms	6.7 ms
Update Query	N\A	1.35 ms	2.49 ms
MIN\MAX Query	17.03 ms	25.90 ms	36.78 ms
Range Query	30.36 ms	55 ms	36.76 ms
Nested Query	22.11 ms	31.71 ms	17.61 ms
GROUP BY Query	29.24 ms	34.58 ms	19.97 ms
Join	648 ms	2861 ms	60.018 ms

Table 4.1: *Summarization of Query Performances on SQL and MongoDB*

4.4 Full Text Index (FTI)

As discussed earlier, one of the features of this application is to enable a user to search for a business based on a keyword. For this functionality, we will create an FTI (Full Text Index) on the reviews. To create an FTI in SQL I have used Lucene search engine library, while for MongoDB I used an inbuilt utility.

4.4.1 Creating an FTI for a SQL Table

Following are the steps to create an FTI on a table using Lucene search engine library.

- Create a directory which will be used by the Lucene search engine to create and store the index. The FTI which is created is stored externally in a folder.

```
string indexFileLocation =
@"C:\User\ankit\document\project\luceneIndex";
```

```
Lucene.Net.Store.Directory dir =
Lucene.Net.Store.FSDirectory.GetDirectory(indexFileLocation, true);
```

- Create an analyzer to process the data from your table. “Lucene.Net.Analysis.Analyzer” class is used to create the analyzer object.
- Create IndexWriter object to write the index to the earlier specified directory. This object takes the directory location and the analyzer object as arguments.

```
Lucene.Net.Index.IndexWriter writer = new
Lucene.Net.Index.IndexWriter(dir, analyzer);
```

- The “writer.AddDocument(reviewData)” (reviewData is one record from the “Review_text” table) command is used to update the FTI in earlier defined directory. The writer.AddDocument method needs to be called for all the records in the table.

Once the index is built, object of “Lucene.Net.Search.Query” class needs to be created with user input “keyword” as argument. This class has a search function which will return all the records in which the keyword was found in the FTI. The average time taken to search for a keyword in Lucene search engine was 1643 ms. As the FTI was created on the review table, we needed to find out distinct pairs of business_id and location_id (to uniquely identify location of a business). This duplicate removal and finding information (e.g., name, address, etc,.) for all the unique businesses took 7648 ms on the average. The whole process took 9291 ms. Every time a new review is written, we need to add that review to the FTI. This process involves analyzing the new review and then, with the help of IndexWriter (discussed earlier) updating the FTI. On the average updating the FTI with Lucene search engine takes about 274 ms. We should also note that MongoDB does not update the FTI if we insert a new document in the collection. In order to include the newly added document we need to

drop the index and build it again. On the average it takes 136 seconds to create an FTI on the review collection.

4.4.2 Creating an FTI on MongoDB Collection

MongoDB has an inbuilt command to create a text index on one or more fields of a collection. The "ensureIndex" command takes the fields as arguments and then creates an FTI on it.

```
//The first argument in BasicDBObject is the field name and the second argume
//indicates the FTI that has to be created.
        db.review.ensureIndex(new BasicDBObject("text", "text")));
```

Once the FTI is created you can query the index using "$search" and "$text" (predefined MongoDB attributes). "$search" should hold the keyword to be searched and this in turn is to be assigned to the "$text" attribute. These two attributes should be passed as arguments to the "find" function. This function will return all the documents where the keyword was found. Below is the code to search for a keyword in FTI.

```
        DBCollection coll = db.getCollection("review");
        BasicDBObject search = new BasicDBObject("$search", "Keyword");
        BasicDBObject textSearch = new BasicDBObject("$text", search);
        DBCursor cursor= coll.find(textSearch);
```

Figure 4.29 shows that the time taken to search for a keyword on the FTI took 7.754 ms. But this search results in duplicate businesses. Hence, to remove duplicates we need to spend another 4176 ms (figure 4.30 shows time taken to remove duplicate business). This problem can be solved if I can embed all the review JSONs related to a particular business within that business JSON.

Figure 4.31 shows the average query execution time to search for a keyword in the new embedded business JSON object. An interesting thing to observe here is that the execution

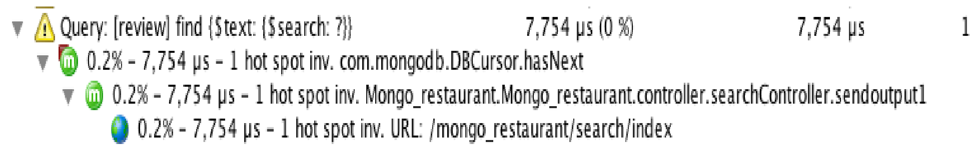

Figure 4.29: *Searching for a keyword in an FTI created on review JSONs*

Figure 4.30: *Finding unique businesses from the output of FTI search*

Query: [business_review] count {$text: {$search: ?}} 893 µs (0 %)
0.0% - 893 µs - 1 hot spot inv. com.mongodb.DBCursor.count
0.0% - 893 µs - 1 hot spot inv. Mongo_restaurant.Mongo_restaurant.controller
0.0% - 893 µs - 1 hot spot inv. URL: /mongo_restaurant/search/index

Figure 4.31: *Searching for a keyword in an FTI created on reviews embedded business JSONs*

time to find a keyword in the review FTI is way higher than to find a keyword in the reviews embedded business JSON. This is because after embedding all the reviews objects related to a business within the respective business JSON, the number of documents has been drastically reduced (we have 300000 review JSONs and only 13490 business JSONs), thus reducing the search time as well.

Chapter 5

Using the Application

In this section we will see the working of the applications which was developed for this report. Although we have developed two application for this report, we will only discuss one of the application in this section (as both of these application have similar functionalities). Following is the working of the SQL application: Once you run the application the first page that is displayed is the homepage.

Figure 5.1 shows the homepage in the application. You can see that on the right top corner of the home page you have links to the registration and login pages. If the user is not already registered, the user needs to register first. The registration will require the user to choose a username, give user details (e.g., address, ph#) and set a password. Once the user is registered, the user can login using the login page. In Figure 5.2 you can see the login screen, which takes username and password to authenticate the user.

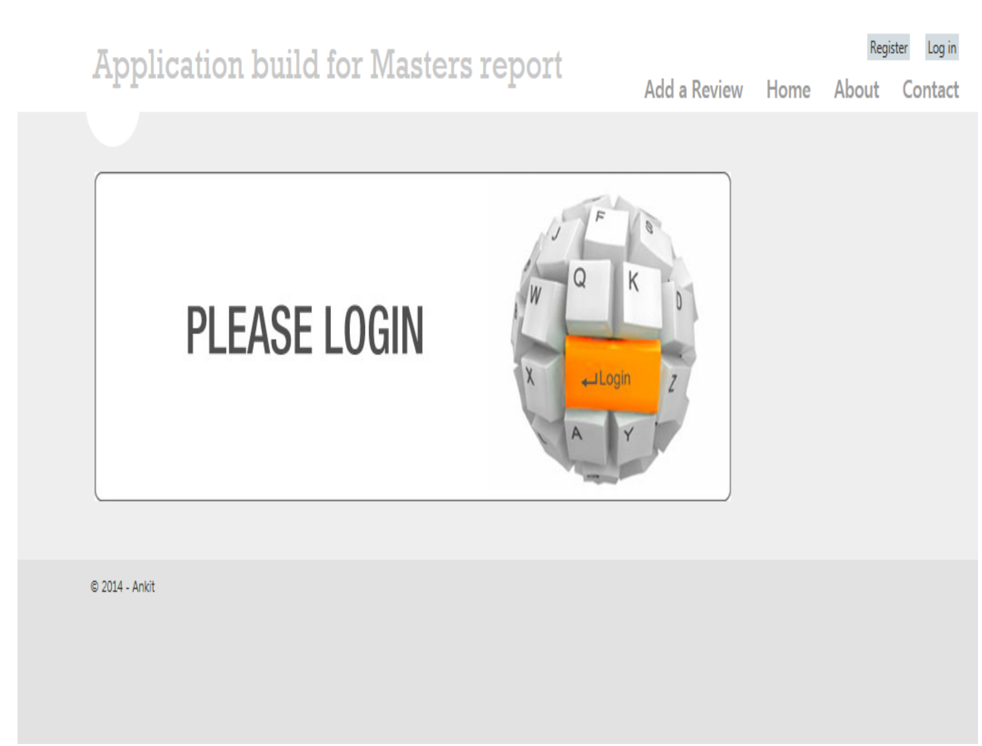

Figure 5.1: *Using the Application: Homepage*

Application build for Masters report

Register Log in

Add a Review Home About Contact

Log in.

Use a local account to log in.

User name

ankit

Password

Remember me?

Log in

Register if you don't have an account.

Use another service to log in.

There are no external authentication services configured. See this article for details on setting up this ASP.NET application to support logging in via external services.

Figure 5.2: *Using the Application: Login page - The JavaScript in this page has an Antiforgery token which makes sure that the user password is protected. Also, the JavaScript does basic validations (e.g., checking the length of the password) to reduce the load on the server.*

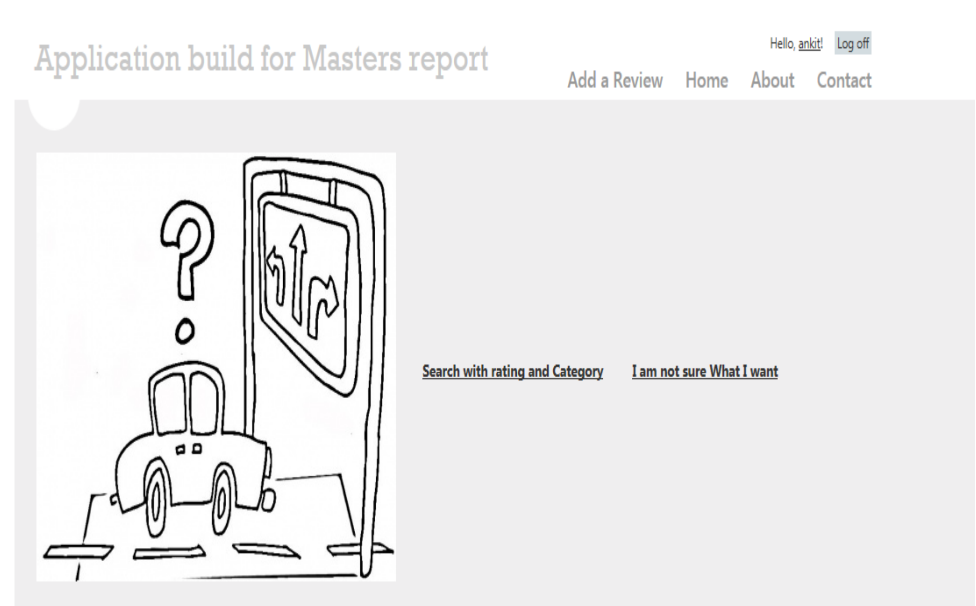

Figure 5.3: *Using the Application: User homepage*

Once the user is authenticated, the user is redirected to the user homepage. As you can see in Figure 5.3, the user has two ways to search for a business. One way is to search for a business by “rating” and “category”, and the other is to search for a business by a keyword.

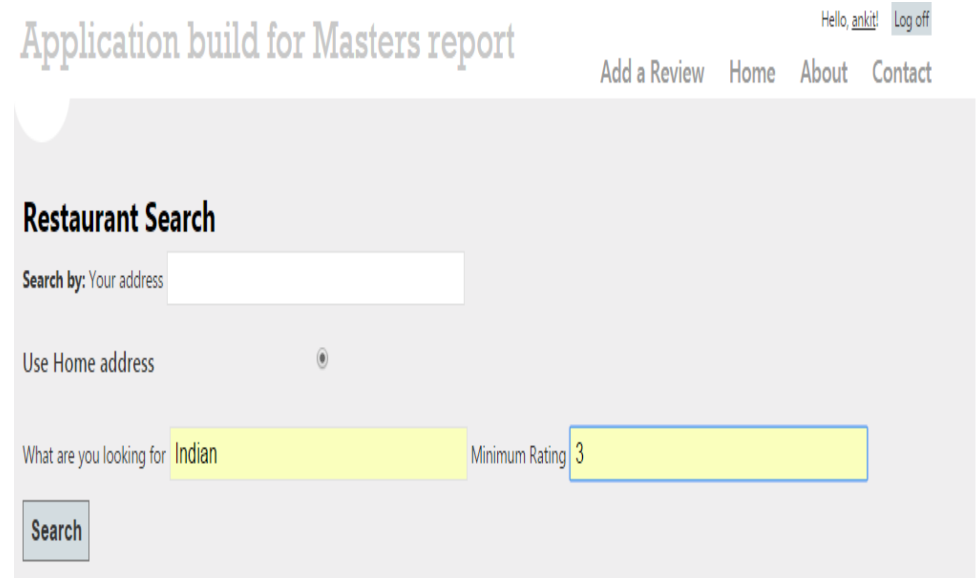

Figure 5.4: *Using the Application: Search page (Search using category and rating)*

Figure 5.4 shows the search page which is used to search for a business using rating and category. Here the user has an option to either use the home address to specify the users current location or enter an address manually. This address is used to calculate the distance between the business and the user location.

Figure 5.5 shows the output of the search. The output has two parts: one is the graph and other is the map. The graph has a center node which represents the user (current location of the user) and this center node is connected to business nodes (all the other nodes) with the help of edges. Here the size of the business nodes signifies the rating (for example a business with rating of 5 would be represented by a bigger node than a business with 3 rating) and the length of the edge signifies the distance between the user and the

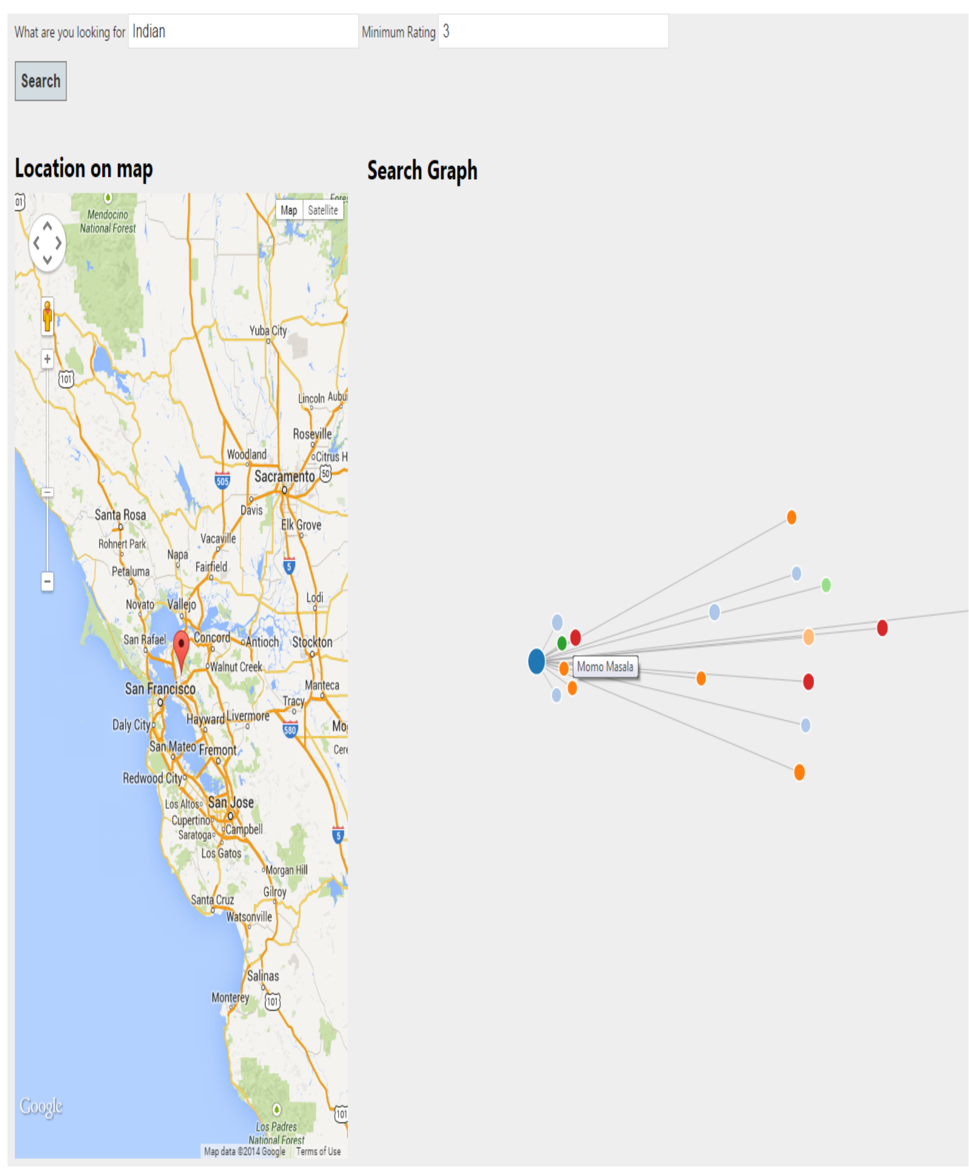

Figure 5.5: *Using the Application: Output of the search - The graph is generated using d3.js and the map is generated with the help of google map API.*

business (the longer the edge, the farther the business). Once the user hovers over a node, the map section of the output screen shows its positions dynamically. If the user wishes to write a review for a particular business, the user needs to click on the node. Figure 5.6 shows the admin homepage. All the users with admin role can add, remove and edit a business or a user. This page gives admin the access to review all the users and businesses.

Welcome Admin

Review All Users

Review All Business

Figure 5.6: *Using the Application: Admin homepage*

If admin clicks on the "Review all business" link, the admin will be redirected to the page shown in Figure 5.7. Here the admin can search for a business to review. Also, the admin is given the option to create a new business, delete the searched business, edit the searched business or see the details of the searched business (Edit, Delete and Details links are listed just on the left side of the business which is searched). Figures 5.8 and 5.9 show

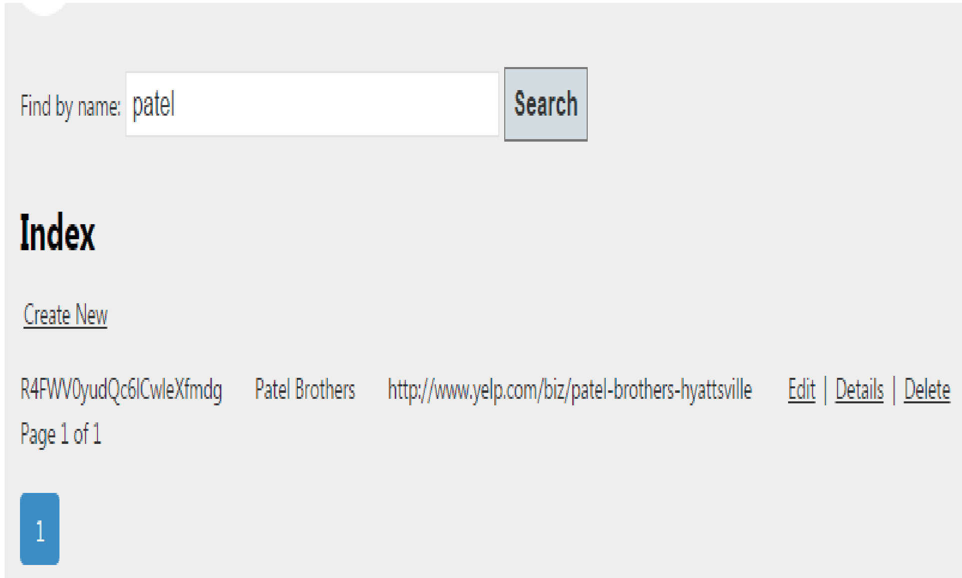

Figure 5.7: *Using the Application: Business homepage*

the pages to create and edit a business. Similar pages also are available to add, delete and review all the existing users.

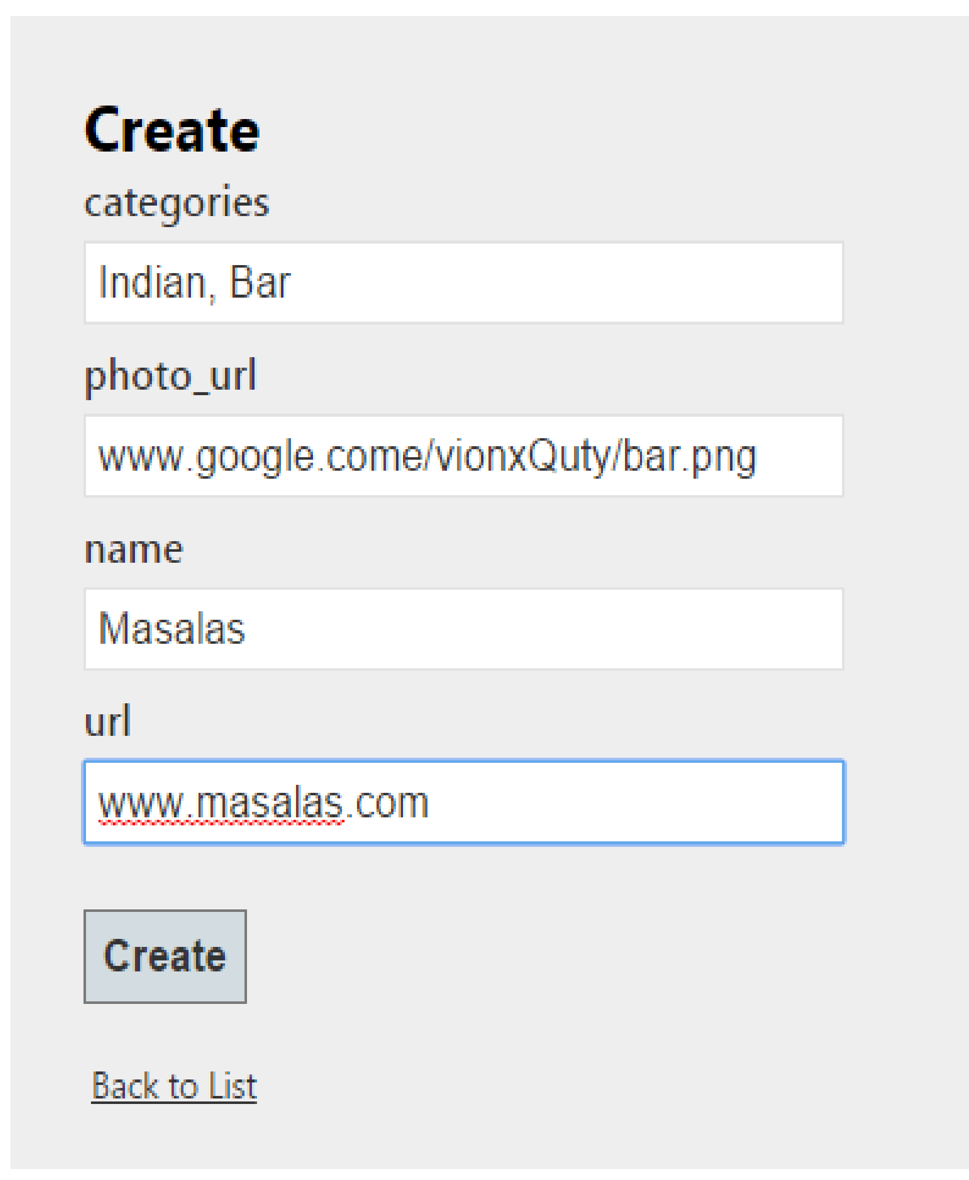

Figure 5.8: *Using the Application: Create a business page*

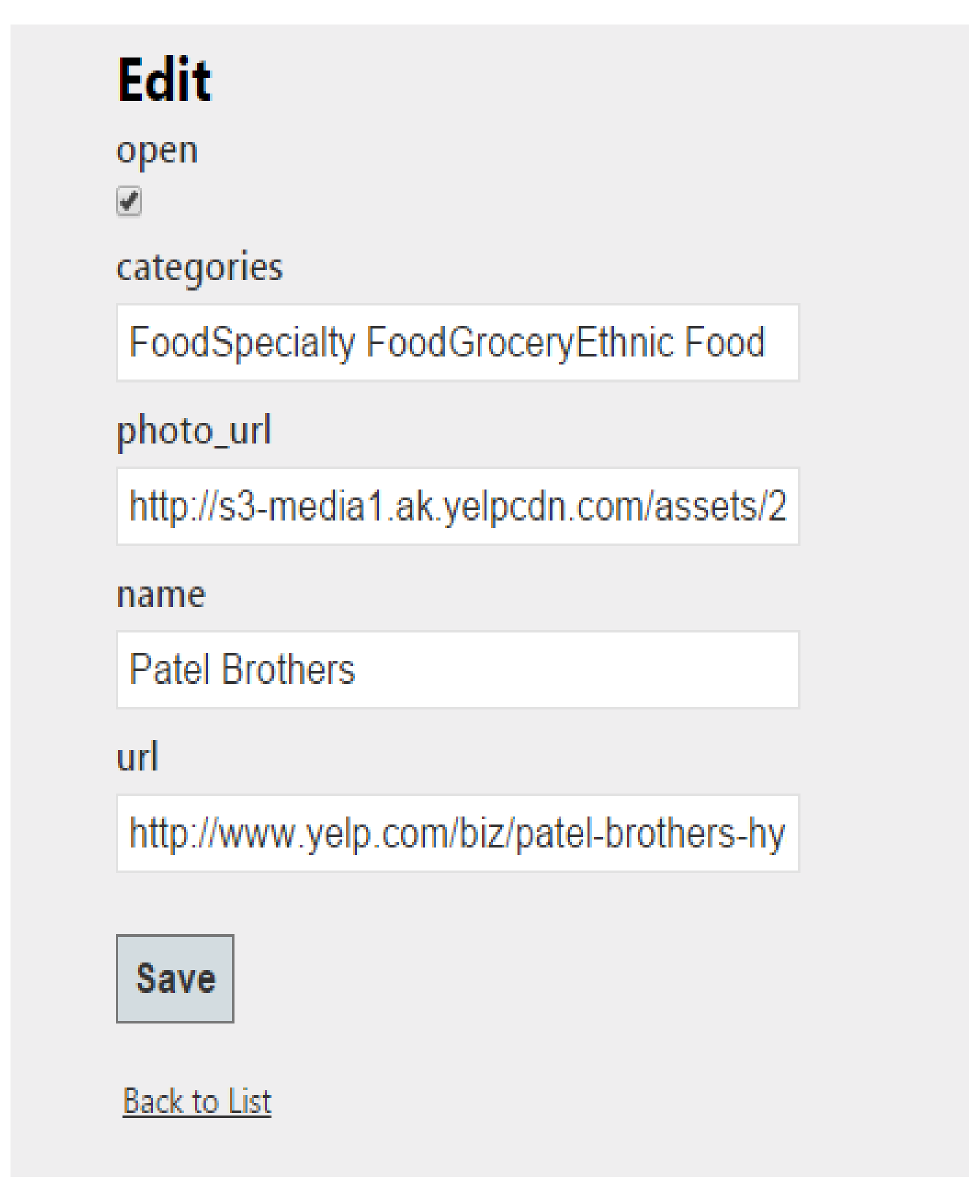

Figure 5.9: *Using the Application: Edit a business page*

Chapter 6

Testing

For testing this application, I am going to use web performance testing. Web performance tests are included in load tests to measure the performance of the web application under the stress of multiple users. The web performance test is recorded by browsing a website as an end user. As you move through the site, requests are recorded and added to the test in Visual Studio Ultimate. After you finish recording, you can customize the test by editing its properties. We will create two basic web performance tests: one as admin_web, which will have requests to all admin related pages and other as user_web which will have request to all user specific pages. In the next step, we will create a load test to note the performance of the application by simulating different factors (e.g., the number of users, using different browsers and different connection speeds).

6.1 Web Performance Testing

As discussed before, in these application we have two user types: Admin and User. We will create two web performance tests, one for each of these roles. In each of these web performance tests, we will test the respective web pages. Web performance tests are very

effective to test a web application because it not only records requests and response times, but also gives a detailed description of the components within the web page. This description can be used to improve the application. For example, in this application some of the web pages took over 3 seconds to load, as seen by inspecting the web performance test results. I found out that the delay was due to loading JavaScript and Ajax library from the web. To bring down the response time, I downloaded the Ajax library and made the application to load the local copy.

Request	Status	Total Time
http://localhost:4842/	200 OK	0.051 sec
http://localhost:4842/Account/Login	200 OK	0.113 sec
http://localhost:4842/Restaurant	200 OK	0.041 sec
http://localhost:4842/index	200 OK	0.068 sec
http://localhost:4842/Home/Contact	200 OK	0.065 sec
http://localhost:4842/Home/About	200 OK	0.083 sec

Figure 6.1: *Performance testing (user_web): Webpages related to the user role and their average response times*

http://localhost:4842/	200 OK	0.057 sec
http://localhost:4842/Home/admin	200 OK	0.068 sec
http://localhost:4842/Restaurant/List	200 OK	0.349 sec
http://localhost:4842/Restaurant/Edit/rwuMWGqt7L5I5IDsOYv1dw	200 OK	0.201 sec
http://localhost:4842/Restaurant/Details/rwuMWGqt7L5I5IDsOYv1dw	200 OK	0.221 sec
http://localhost:4842/Restaurant/Delete/rwuMWGqt7L5I5IDsOYv1dw	200 OK	0.175 sec

Figure 6.2: *Performance testing (admin_web): Webpages related to the admin role and their average response times*

In Figure 6.1, you can see the response time of all the web pages related to the user role. The response time ranges from 51 ms to 113 ms, which is very good. We will include this test as part of a load test and then see if this response time is maintained when the number of users increases. Figure 6.2 shows the response time of all the web pages related to the admin role. The response time varies from 57 ms to 349 ms.

6.2 Load Testing

The primary purpose of load tests is to simulate many users accessing the server at the same time. I will create a load test to simulate 50 to 1000 users to run the above mentioned web performance test 10 to 100 times within 10 seconds to 3 minutes. I will also add the simulation parameters to execute the test with different user inputs.

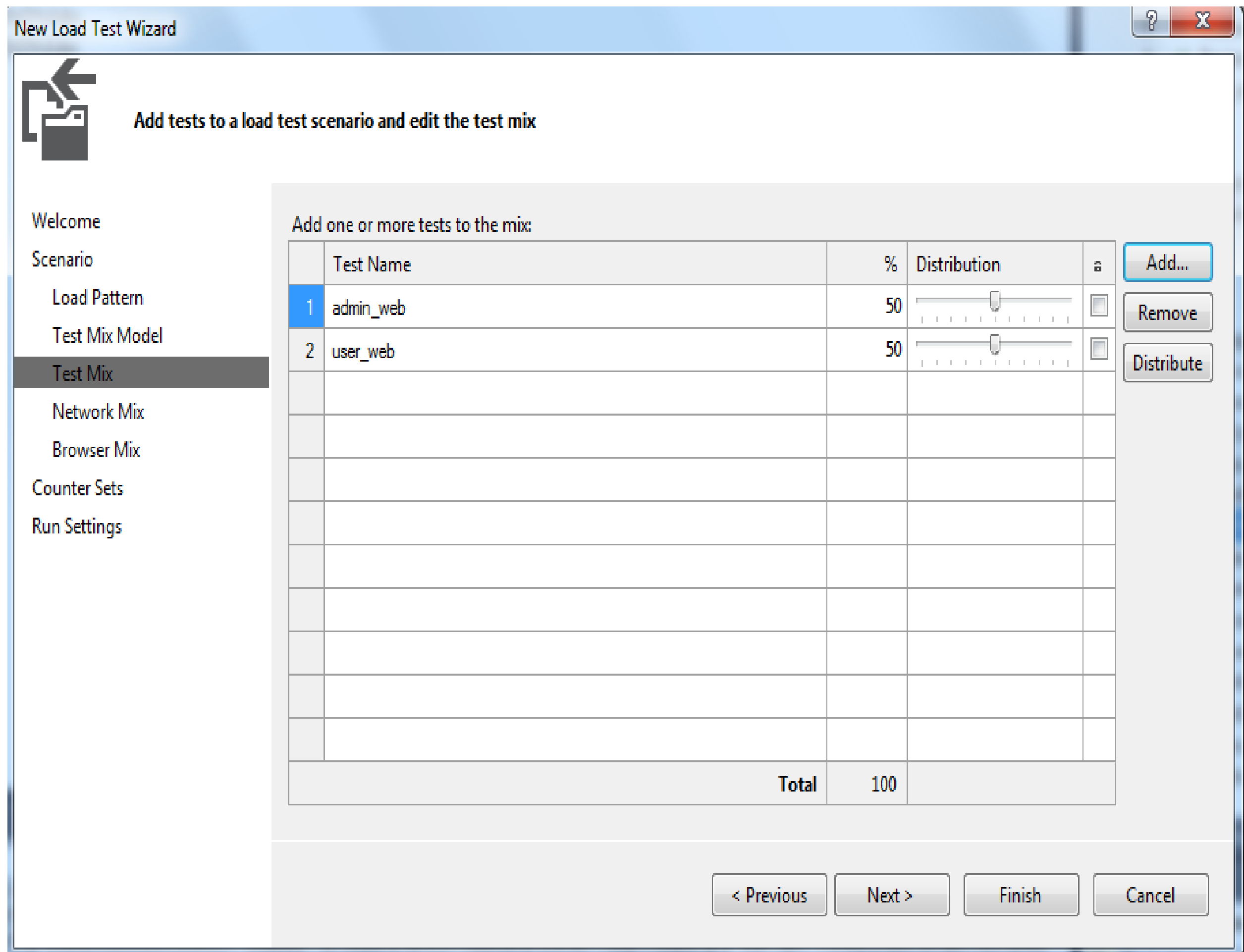

Figure 6.3: *Load test (user_admin): Including web performance tests*

- Figure 6.3 shows that we have included the web performance tests created earlier (user_web and admin_web). Also, we set the number of users accessing the webpages and also the distribution of these users to each of the web performance tests.

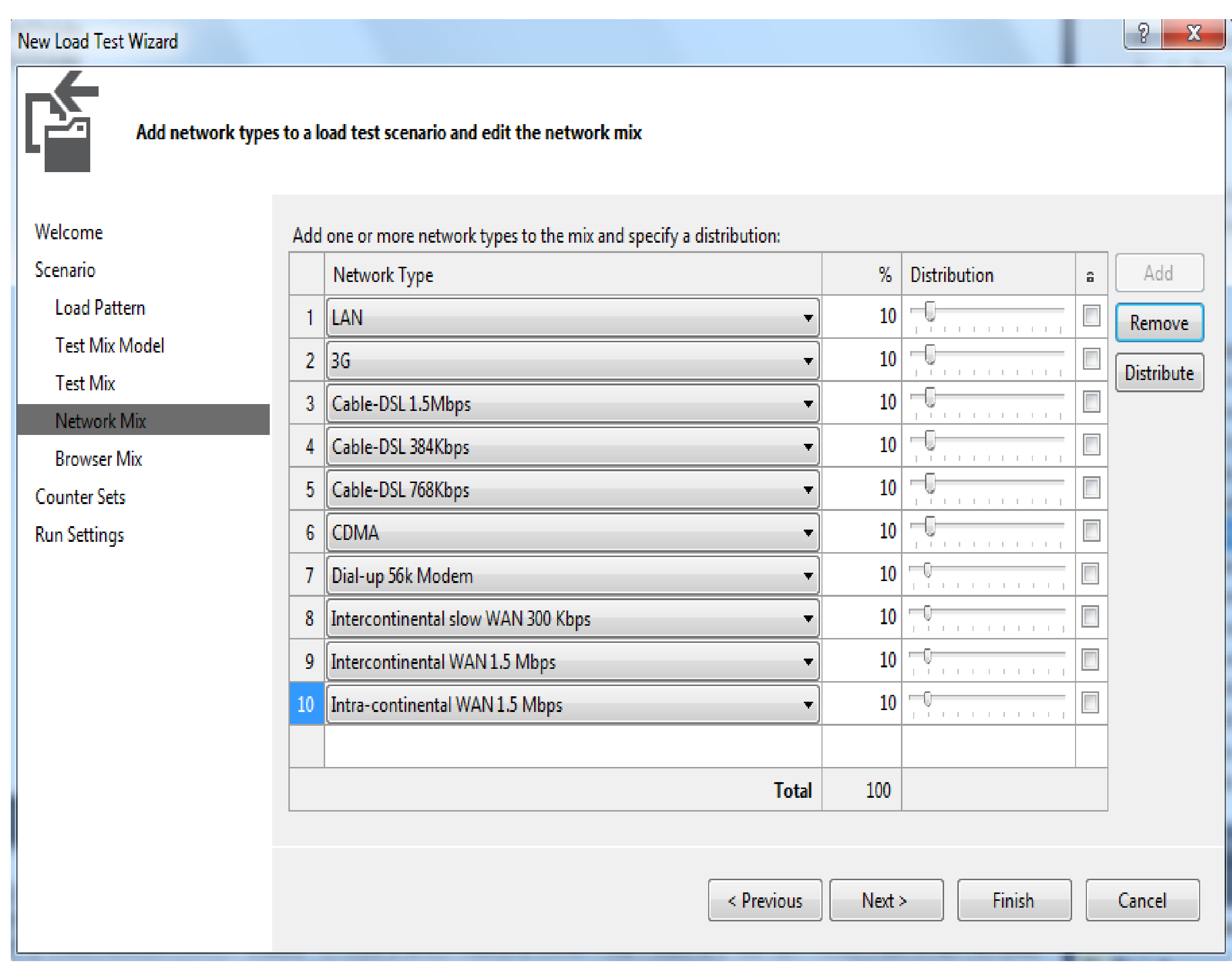

Figure 6.4: *Load test (user_admin): Connection speeds included*

- Once we have added the web performance test and set the distribution, we set the various types of connections we wanted to simulate the test with. Figure 6.4 shows the various connections I have included in this test.

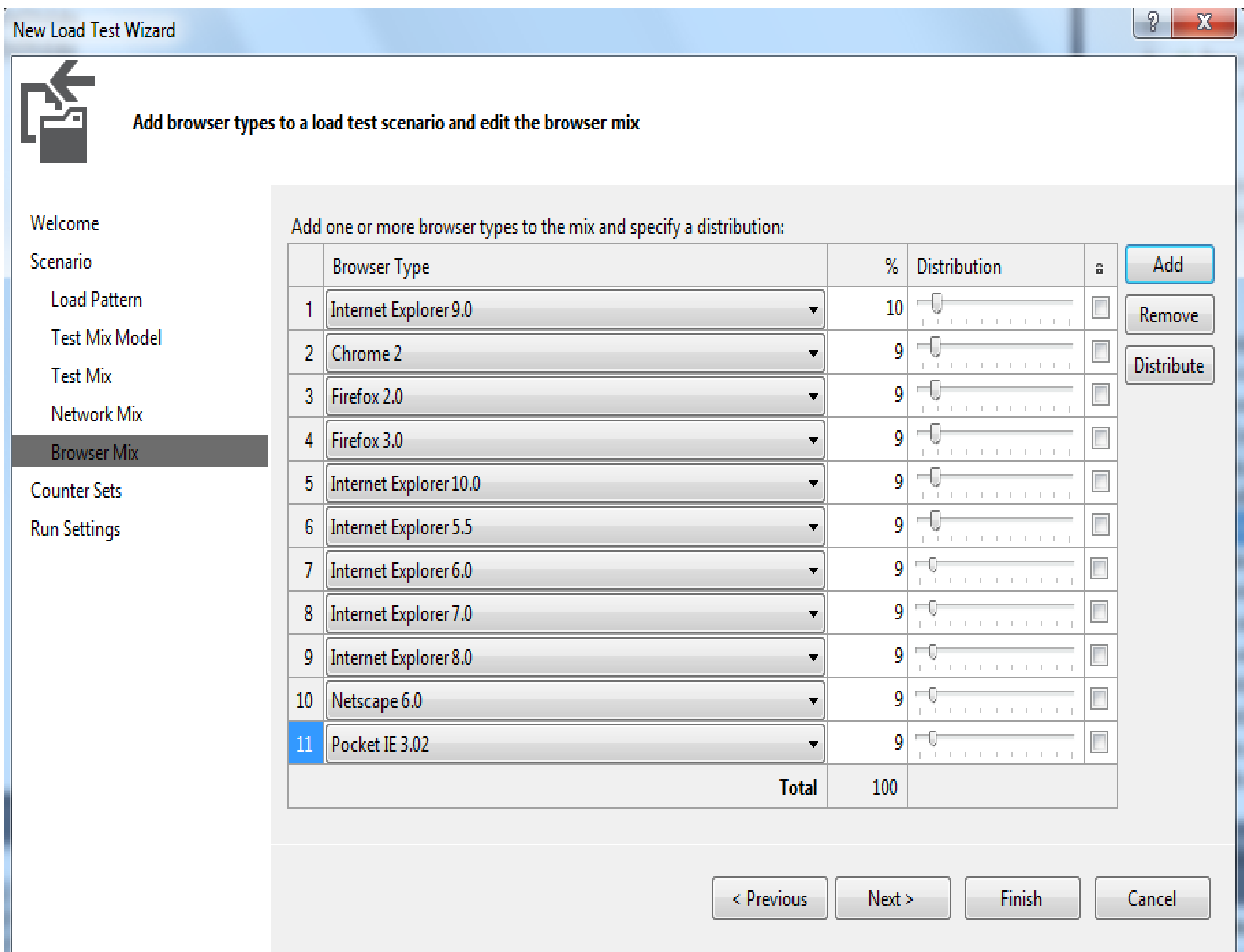

Figure 6.5: *Load test (user_admin): Browsers included in the load test*

- The next step is to make sure the application performs well on different browsers. Figure 6.5 shows the various browsers I have added in the load test.

▾ **Page Results**

URL (Link to More Details)	Scenario	Test	Avg. Page Time (sec)	Count
http://localhost:4842/Restaurant/Delete/rwuMWGqt7L5l5lDsOYv1dw	user_admin	admin_web	4.15	2,014
http://localhost:4842/Account/Login	user_admin	user_web	4.15	2,503
http://localhost:4842/Restaurant/Details/rwuMWGqt7L5l5lDsOYv1dw	user_admin	admin_web	4.14	2,078
http://localhost:4842/Home/About	user_admin	user_web	4.14	1,991
http://localhost:4842/Home/Contact	user_admin	user_web	4.14	2,059
http://localhost:4842/Home/admin	user_admin	admin_web	4.14	2,497
http://localhost:4842/index	user_admin	user_web	4.12	2,503
http://localhost:4842/Restaurant/Edit/rwuMWGqt7L5l5lDsOYv1dw	user_admin	admin_web	4.12	2,495
http://localhost:4842/Restaurant/List	user_admin	admin_web	3.75	2,497
http://localhost:4842/	user_admin	admin_web	3.73	2,497
http://localhost:4842/Restaurant	user_admin	user_web	3.72	2,503
http://localhost:4842/	user_admin	user_web	3.70	2,503

Figure 6.6: *Load test (user_admin): The result of the load test*

Figure 6.6 shows the result of the load test. You can see that on the average, every web page was accessed about 2500 times by 1000 users in 3 minutes. There were no errors nor denial of service. All the request were completed with a proper response. This shows that the application works well with a variety of browsers and connection speeds.

Chapter 7

Technologies Used

Following are the technologies used in the development of the SQL application.

- **ASP .NET MVC 4:** It is a framework for building standard and scalable web applications. This framework makes use of the MVC pattern (discussed in Section 1.1).
- **C#:** It is an object oriented programming language which was developed by Microsoft. It is a general purpose language which has proven to be very efficient to develop web applications.
- **Java Script:** Java Script is a programming language for the web, used to create dynamic web pages. In this project most of the views are written in Java Script. These scripts are responsible for features such as user input validation (e.g., checking length of password).
- **SQLite:** "SQLite is a software library that implements a self-contained, server less, zero-configuration, transactional SQL database engine" [9]. ASP.NET framework has SQLite integrated with it, hence we need not install any software explicitly.
- **LINQ:** LINQ stands for Language-Integrated Query. Unlike traditional queries, where query output is expressed as simple strings without any type checking at compile time,

LINQ queries are written against strongly typed objects which hold the output of the query without any data loss (these objects match the table structure). It is also easier to use these objects in our code.

- **Razor View Engine:** This technology comes integrated with the ASP.NET MVC 4. Razor is not a client side technology, it generates views within the application server. Once these views are generated, they are used on client systems. The process of converting Razor syntax to html code happens during compilation of the application. Razor syntax is very similar to any modern day object oriented general purpose language, which makes it very easy to learn and use.

- **MiniProfiler:** In this report, we need to compare query execution time for SQL and MongoDB. For finding the execution time for SQL statements, I have used "MiniProfiler". This software is exclusively developed for .NET framework and has capability to segregate (from other controller logic) and profile only SQL statements.

Following are the technologies used in the development of the MongoDB application.

- **Java Spring:** The Java Spring is an open source application development framework, built on Java platform. I have used the Java Spring web module to leverage the MVC pattern. I have also used Java Spring Security module to implement the user role and authentication functionality.

- **Java:** It is an object oriented programming language. I have used this language to write the controller login in my application.

- **MongoDB database server:** It is an open source document database.

- **JSP and HTML:** These technologies are used to create client side web pages.

Technology	Lines of code
C#	1568
Java	864
JavaScript	430
Razor view engine	630
HTML & CSS	760
SQL Queries	118
MongoDB Queries	76
Junit	247
XML	386

Table 7.1: *Project Metrics*

- **JProfiler:** In this report we need to compare query execution time for SQL and MongoDB. For finding the execution time for MongoDB statements, I have used "JProfiler". This software is exclusively developed for Java Spring framework and has capability to segregate (from other controller logic) and profile only MongoDB statements.

Other than the above listed technologies, I have used the following APIs and Java Script libraries.

- Google Maps API: To generate maps.
- Google Distance Matrix API: To calculate distance between the user and businesses.
- d3.js (Java Script Library) - Force graphs: To generate graphs for the search output.

Table 7.1 shows the project metrics. This includes the technology name and the corresponding number of lines of code written using the corresponding technology.

Chapter 8

Lessons Learned and Conclusions

After comparing and contrasting the SQL databases with the NoSQL databases, we have learned that both these databases have their own set of pros and cons. The decision to use either the SQL database or the MongoDB database as a backend needs to taken based on developer's requirements. Following are the factors I came up with during the course of this report, which separates SQL databases from NoSQL databases.

- Data Modeling: In SQL databases to avoid anomalies and data redundancy, we need to normalize data before storing. Normalization would cause the data to be split into different tables. If we need to access information from more than one table we need to use joins. Joins are expensive operations and would make the query execution slower. On the other hand, MongoDB does not support joins, hence we need to model our data in such a way that all the data which needs to be read within a query is kept in the same collection. If we need to collect information from more than one collection, we need to write the logic to join the data on the controller, which would make the query very slow.

- Loading Data: As MongoDB is a document based database, inserting and deleting a document was found to be very fast when compared to inserting/deleting a row in

SQL. This is due to the constraints (primary key, Unique, Maintaining Indexes, etc.) which are imposed on fields when tables are created.

- Reading Data: MongoDB was found to be very fast when we needed to read the entire dataset. But when a single row was to be read, SQL database was faster. Also, indexing the columns which are commonly used in *group by* and *where* clause will further improve the SQL query performance.

- Text Indexing: I used Lucene search engine library to develop FTI in SQL database. For MongoDB, we used a tool which was already available within the database server. The major advantage which SQL database FTI had over the NoSQL was that the index which was built by Lucene search engine was incremental. That means that if we need to add a new entry into the FTI, we can add that entry to the existing index. In MongoDB we need to rebuild the index every time we need to add a new entry. Although the searching a keyword on MongoDB FTI is faster when compared to that of FTI on SQL, it is not a good option to select a MongoDB database if we have too many updations or additions to our data.

- MongoDB database is faster than SQL database for queries where entire or most part of the database needs to be read. While SQL has proven to be faster for point queries (refer to Table 4.1 for query comparison between SQL and MongoDB).

Bibliography

[1] Microsoft Msdn, *http://www.microsoftvirtualacademy.com/training-courses/introduction-to-asp-net-mvc.* Aug,2014

[2] Garcia-Molina, Hector and Ullman, Jeffrey D. and Widom, Jennifer *Database Systems: The Complete Book*, 2008, 9780131873254

[3] *http://databases.about.com/od/specificproducts/a/acid.htm* Accessed in October, 2014

[4] *http://nosql-database.org/* Accessed in October, 2014

[5] *http://robertgreiner.com/2014/06/cap-theorem-explained/* Robert Greiner, June 18, 2014

[6] *http://www.aerospike.com/what-is-a-nosql-key-value-store/* Accessed in October, 2014

[7] *http://www.mongodb.org/about/introduction/* Accessed in October, 2014

[8] *http://www.mongodb.org/about/introduction/* Accessed in October, 2014

[9] *http://www.sqlite.org/* Accessed in August, 2014