**Introduction**

For this NoSQL report, I am using CouchDB which is a document based database technology. CouchDB uses a schema-free design and is designed with a focus on data availability and partition tolerance. The moto of CouchDB is Relax. Unfortunately, since I am just learning this technology, it felt like I was working with OuchDB because the process was relatively painful. Since CouchDB is not very mature, there is not a lot of information on the web about it. The documentation on the CouchDB website is not organized very well which caused me to spend more time than I wanted just searching how to do things. However, as I learned, I gained an appreciation for the simplicity of CouchDB.

**Extract**

Instead of extracting data from MySQL as I have done in previous assignment, I just used the JSON files given to us in the class folder. These data are mostly well-formed JSON files which is what CouchDB expects. You can not use other file formats with CouchDB.

**Transform**

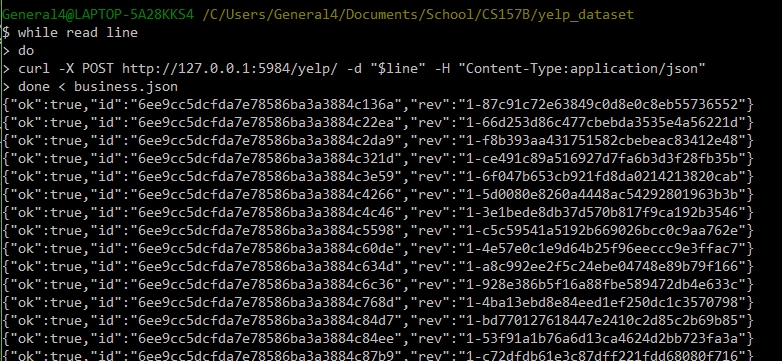
Since it was mentioned numerous times in class during lecture, I thought it would be cool to use Spark to transform the Yelp data before loading it into CouchDB. Unfortunately, I could not find a CouchDB Spark connector. After more thought, it occurred to me that, even if I did transform the data with Spark, the **saveAsTextFile** function does not save the files in the JSON format that CouchDB expects. However, through research I learned that you can use SparkSQL to read-in JSON files and even save the Data Frames as JSON files. If I had more time, I would consider learning SparkSQL and performing transformations this way.

Transforming data with MySQL is another option. Obviously, using MySQL would not be very fast. I decided against using this option for the same reason I did not use Spark. Even though MySQL has added support to query JSON data, I did not see any documentation on saving in the JSON format. The output files of the transformation need to be well formed JSON data. If it is not, I will need to add another step such as writing a script to parse the files and convert them into JSON. This is something I don’t have time to do. Also, it really wouldn’t make much sense in this context any way. In a similar vein, I could skip using Spark or MySQL and just write a Python script to transform the data. If it’s a CGI script, it could be automated to run daily. The script could query MySQL gathering data and form it into JSON to be delivered to CouchDB. None of these options were going to work for this assignment.

Since CouchDB allows you to use JavaScript to manipulate the data, I decided that it has built-in support to transform data and that I did not need to use another technology. However, before I could transform the data, I need to ingest it into a CouchDB database. This would cause redundant data, but in the world of big data this seems to be normal. Data ingestion will be discussed next in the Load section. And I will discuss more about data transformation in the Query #1 section.

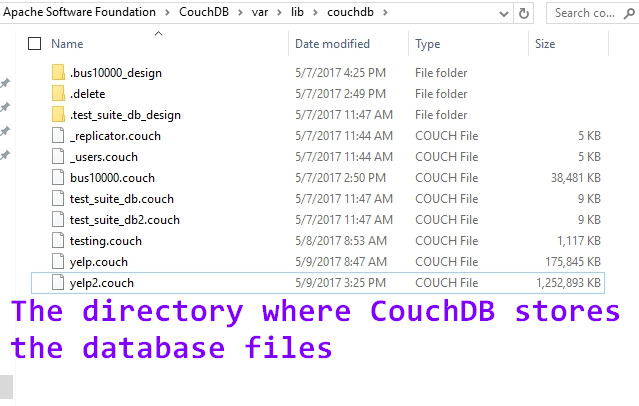
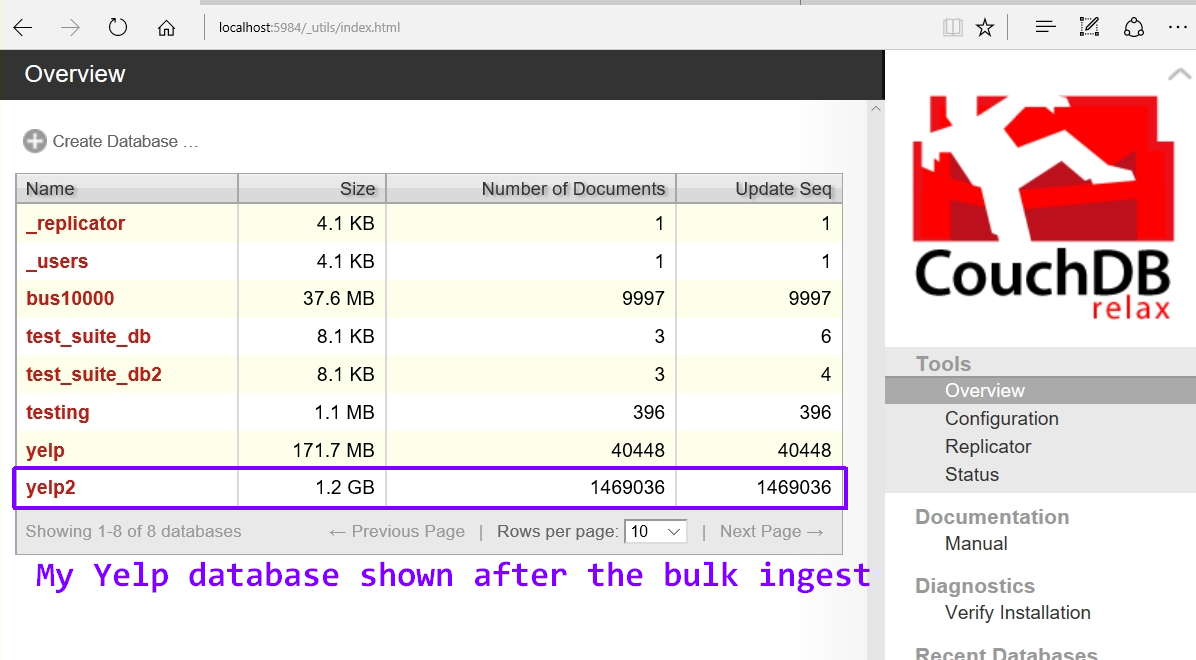
**Load**

As I mentioned above, the documentation for CouchDB is not very good. I initially could not find information on batch loading a set of JSON files into a CouchDB database. Also, when using the CouchDB web interface, there is no upload dialog to import data files. So, my initial data ingestion was less than impressive. I created a database using the CouchDB interface and by using the command line terminal, I wrote a simple script to read lines of the business file and send Curl POST requests to the database. As you can imagine, since this method read each record line-by-line, it was not very fast.



As shown in the screen shot above, CouchDB prepends every record entered into the database with a unique id and revision number. After two and a half hours, only 40,448 records were imported. I aborted the import in search of a better way. Using this method to import all the data would take forever! This method turned out to be a practical exercise on how not to import data.

After talking with a friend, I was informed on how to do bulk data ingestion with CouchDB. He also had a difficult time locating this information. Basically, you do bulk data ingestion very similarly to how I imported data. Two differences to my curl request shown in the screen shot above is – you must append the directory **\_bulk\_docs** to the path of your database and the script should read in the whole file instead of just a line. However, if the file is too large, the curl POST request can time out. To prevent this, I used a Java program that my friend wrote to chop each Yelp JSON file into many smaller files. Each smaller file contained 10,000 records. For example, after running the program on business.json, the output was 15 files, business0.json – business14.json, each with 10,000 records (except the last file). The whole process only took about 40 minutes – this was a huge improvement. The entire Yelp dataset was put into a single database. My database has no document hierarchy. To extract data from a specific dataset, my queries will use the ‘type’ field that all the documents contain. The value of the type will determine if information is extracted from a particular type.

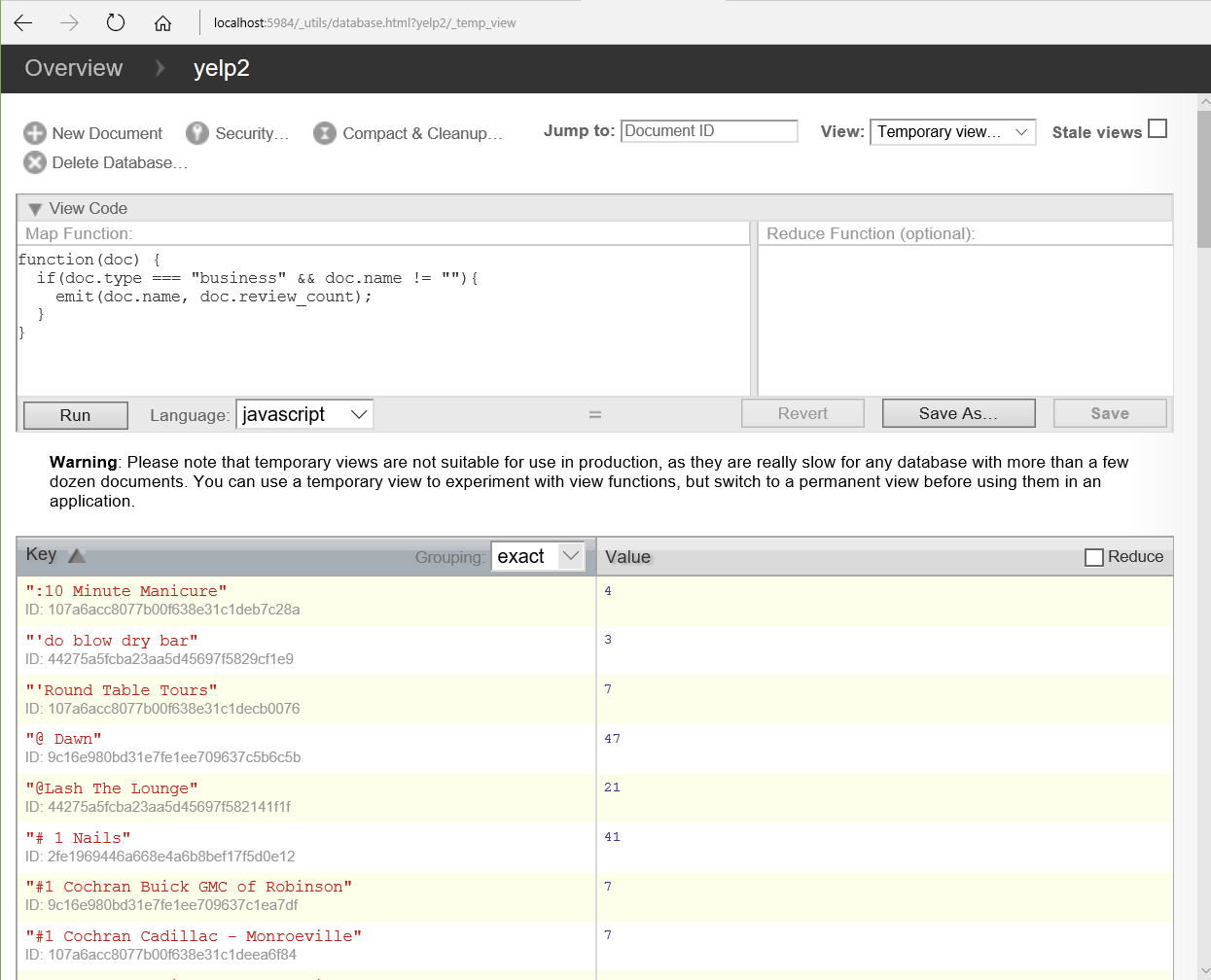


I was impressed with the CouchDB file parser. I know the Yelp data has a lot of data that is not well-formed. Instead of crashing or aborting during ingestion process, CouchDB simply reported when there was bad data and continued with the data ingestion process. This was a refreshing change. With the data finally loaded it was time to try out some querries.

**Query #1**

As I mentioned above, CouchDB allows you to use JavaScript to work with the data. You can write map and reduce functions directly in the CouchDB web interface (called Futon) or you could use the command line. To use the web interface, select **Temporary View** from the interface **View** drop-down menu. Then you can enter a map function using JavaScript to filter the data to get only what you need. My first query was very simple, as I was just getting used to the way CouchDB works:

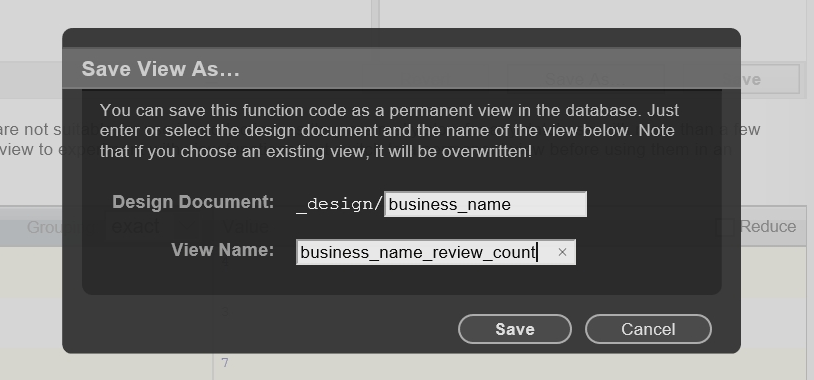
Query: **Select name, review\_count From business;**



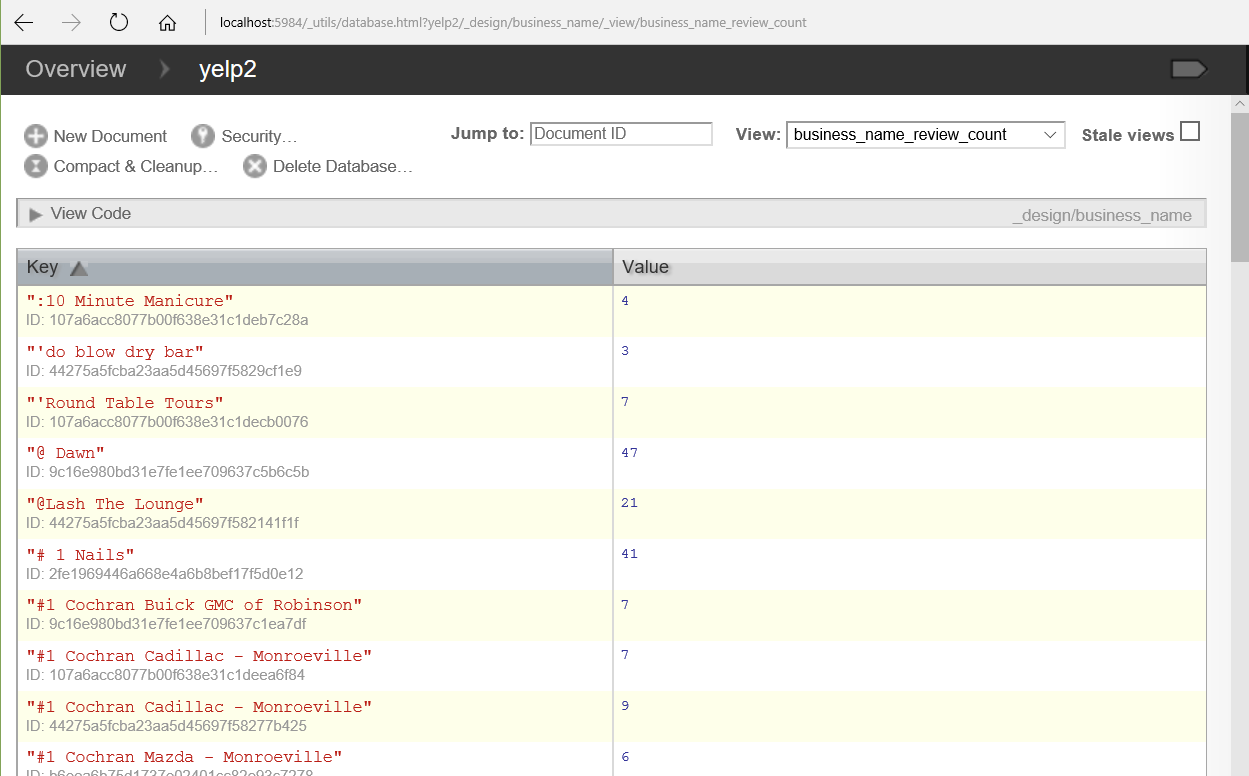
The JavaScript version of this simple query can be seen in the image above. Since CouchDB doesn’t care that some of the documents contain field keys with the same name, you must be specific about which documents you want to extract information from. The *user* dataset contains a field named ‘name’ and another named ‘review\_count’, so in my query I need to specify that I only want information pulled from the *business* dataset. I must leverage the **type** field that each document contains. When I first saw the Yelp data, I wondered why each dataset - business, user, etc… - contained a type field. Now I know. In the context of a document store, this field will help select only the documents of a particular type.

**Data Transformation Revisited**

Since the ouput of the map function produced a subset of the business dataset that might prove to be useful for future querries, I decided to save it as a permanent view. This can be considered a transformation of the business data.



Here is the same info from query #1, but saved as a view. Whenever this information is needed again, it can be pulled up almost instantly since it is permanently stored in the database. Querries that are run against this view will be relatively fast since the whole Yelp dataset is not searched. Views take up disk space, so you don’t want too many of them.



**Query #2**

My second query does the following - for each postal code in Phoenix, find the average star rating for all businesses in that postal code. Here is the MySQL version:

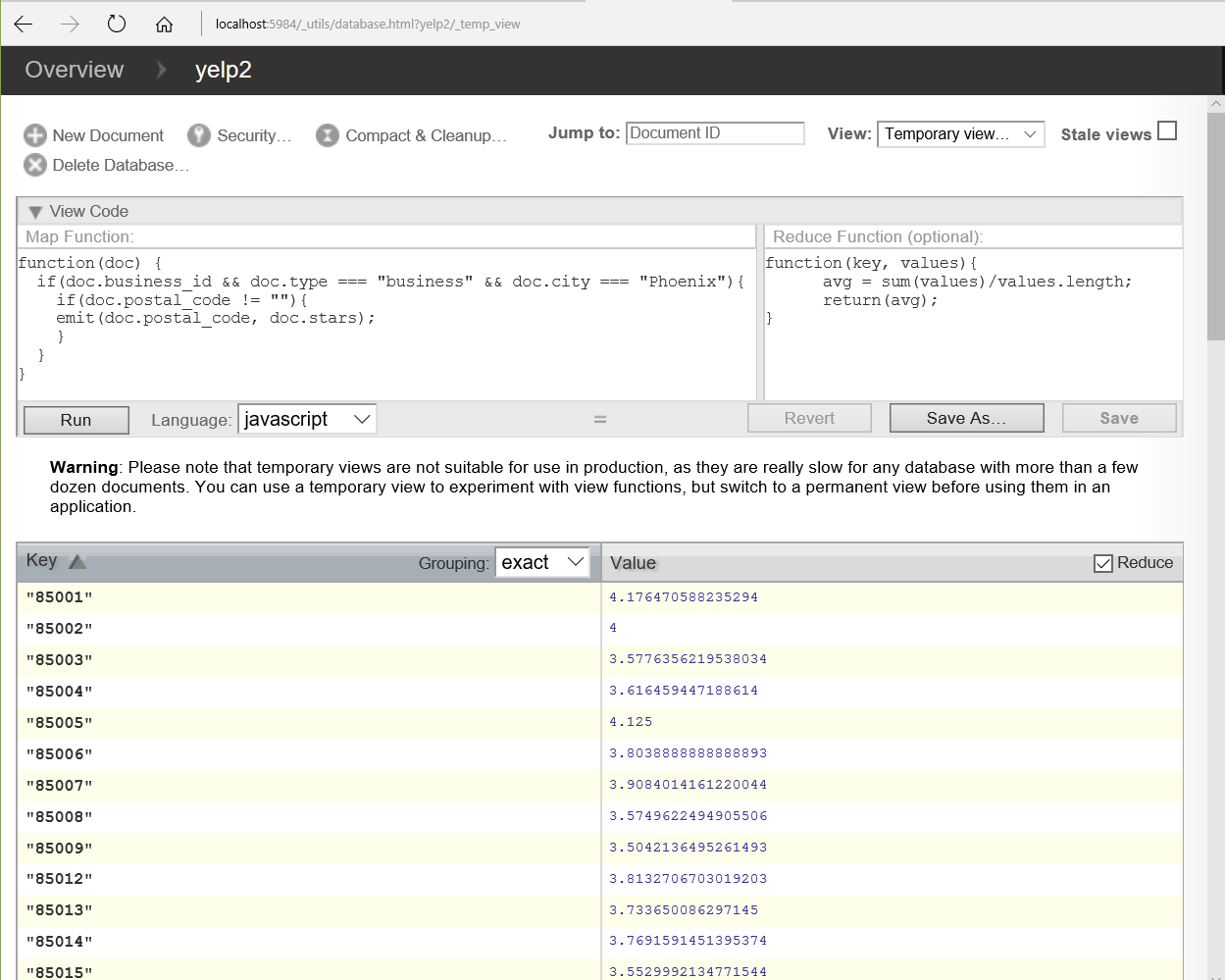
**Select postal\_code, Avg(stars)**

**From business**

**Where city = "Phoenix" And postal\_code != ""**

**Group By postal\_code;**

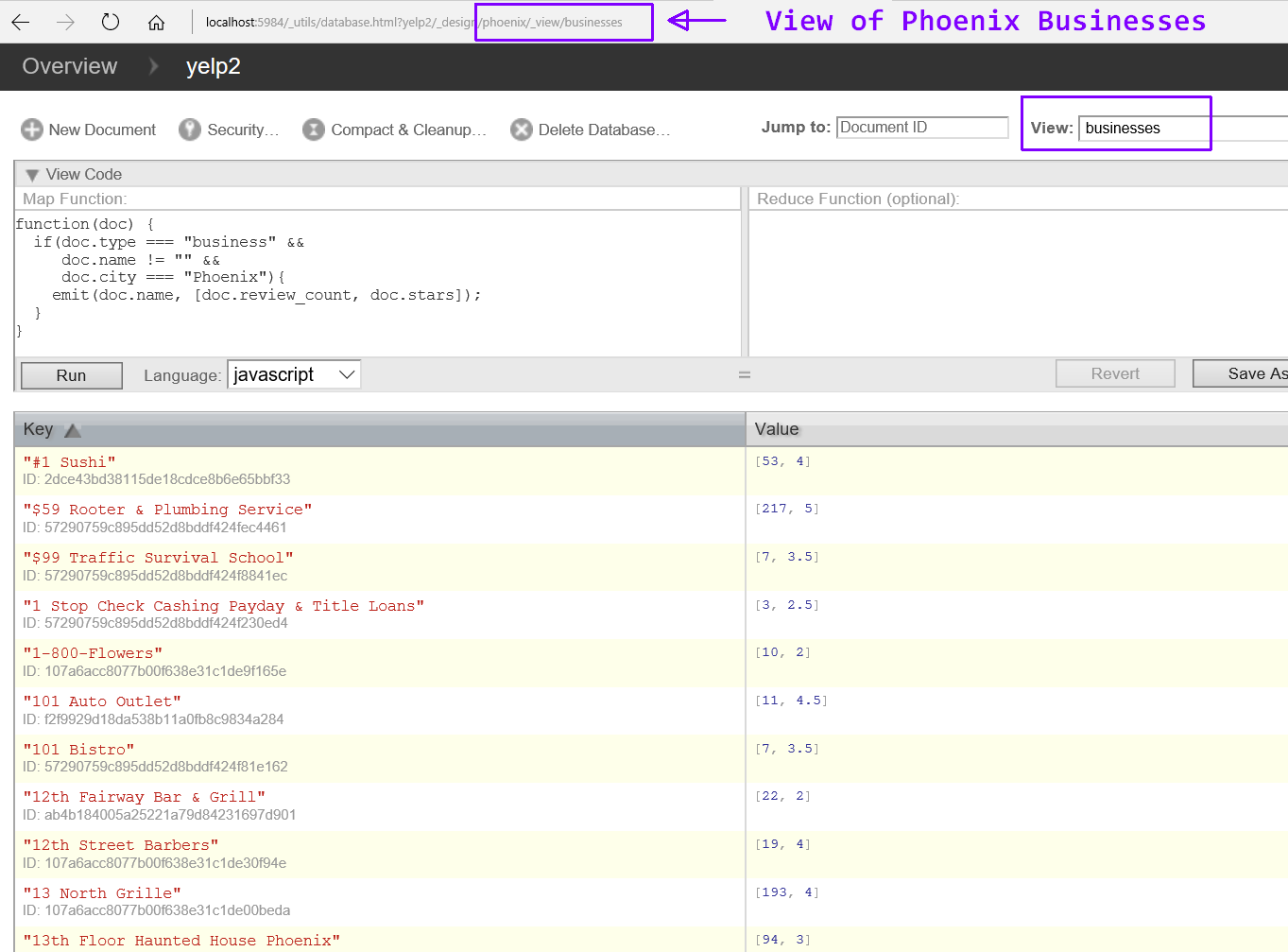
To solve this query, I use a map function and a reduce function. This is very similar to the way I solved a similar query with Hadoop MapReduce or with Spark. The map function selects all postal codes in phoenix and their corresponding star rating. The reduce function is run once for each distinct postal code. To compute an average, it counts the number of star values for each postal code and also sums each set of star values. It’s nice how my code does not need to do any type casting – like from string to float for instance – CouchDB takes care of this behind the scenes.



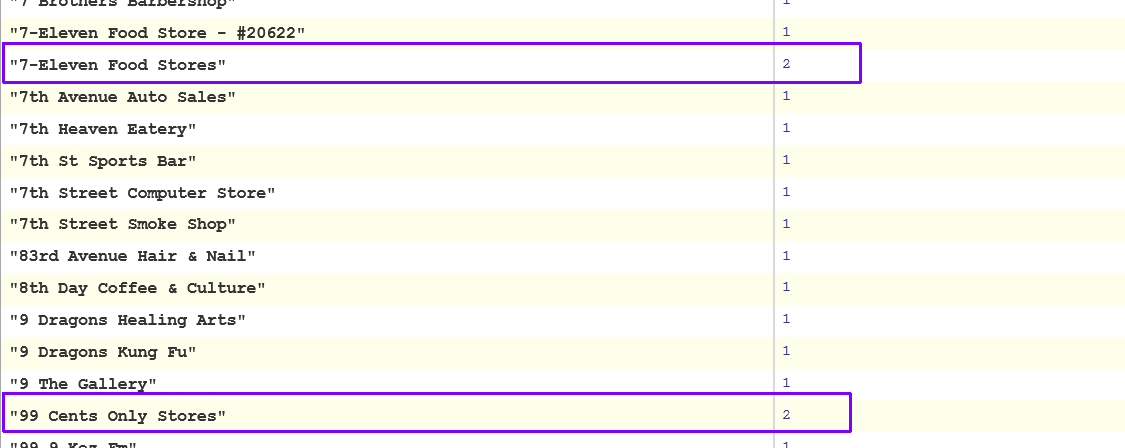
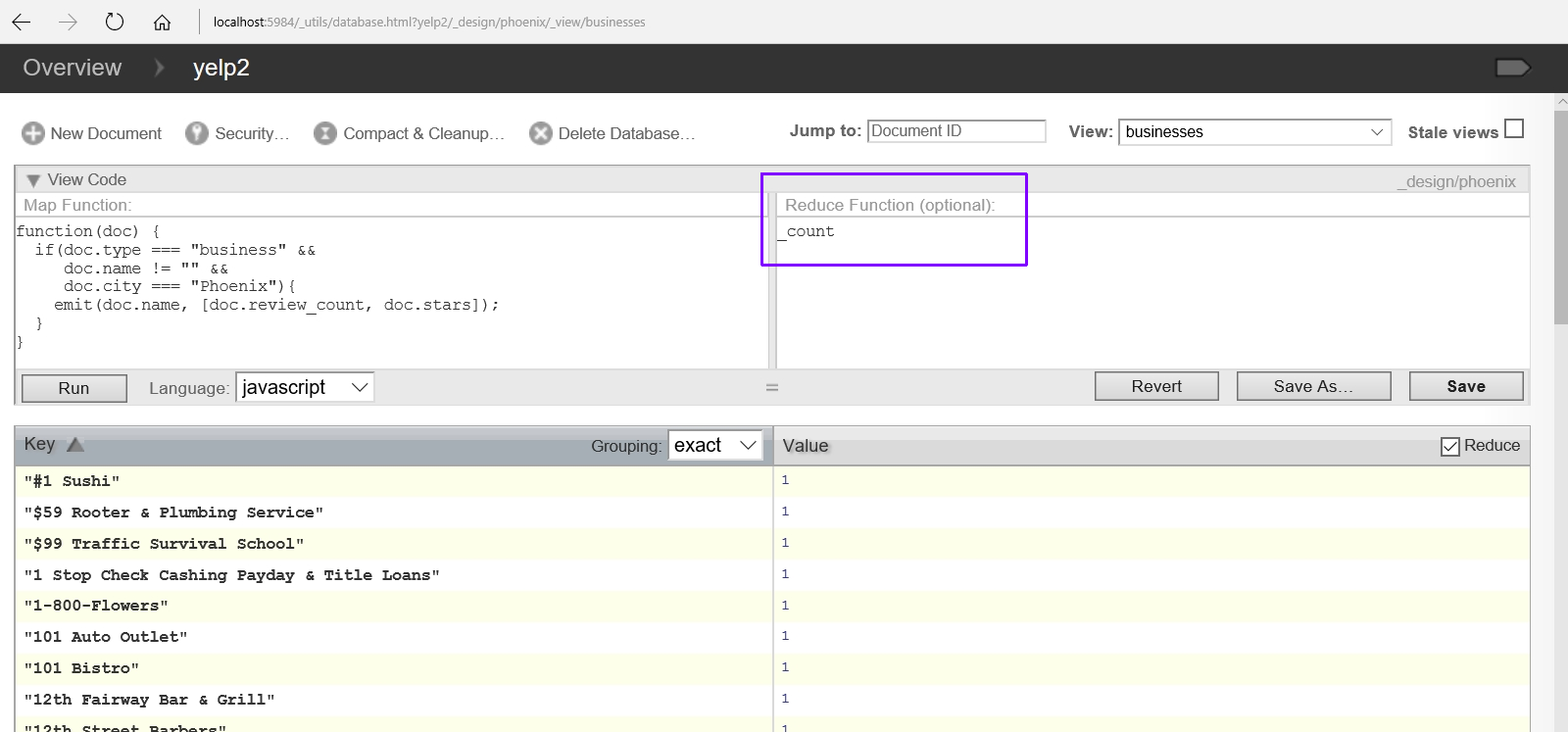
**Query #3**

**Find the count of businesses in Phoenix that share the same name.**

For this query, I created another view first. This view contains the name of Phoenix business as a key and their corresponding review\_count and star rating. The query to create this view took 4 minutes and 40 seconds to complete. In the Optimization section below, I will compare the relative differences in query speeds when running queries over a view versus the entire database. The map function will create objects with a business name as a key and an array which contains the review\_count and stars as the value. To do the same thing as a Hadoop Map function would have taken much more code. I really like how a little CouchDB code does so much.



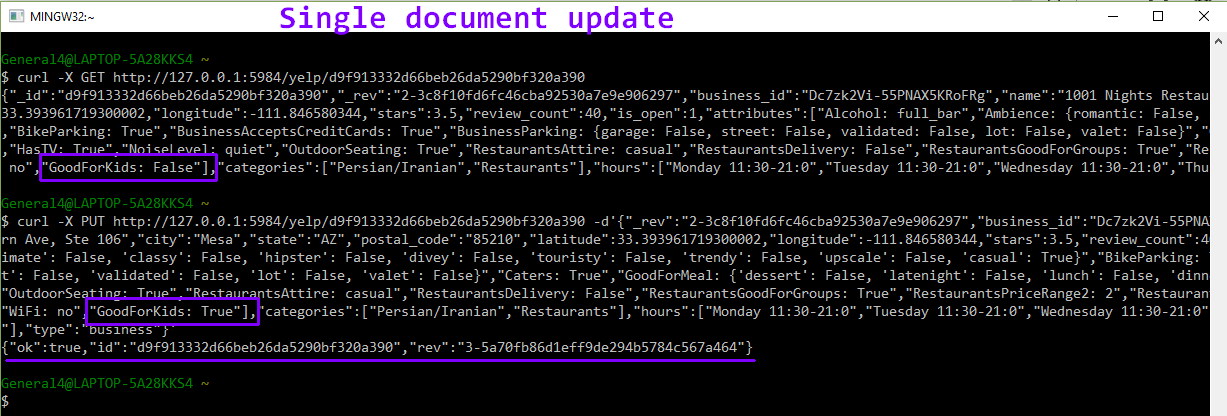
By applying the built-in CouchDB **\_count** function as a reducer, I can determine if any businesses share the same name. CouchDB has other built-in reducer functions as well, such as **\_sum** and **\_stats.**



The image above shows some of the businesses in Phoenix that share the same name. These were duplicate keys that were grouped and the count of each group is diplayed.

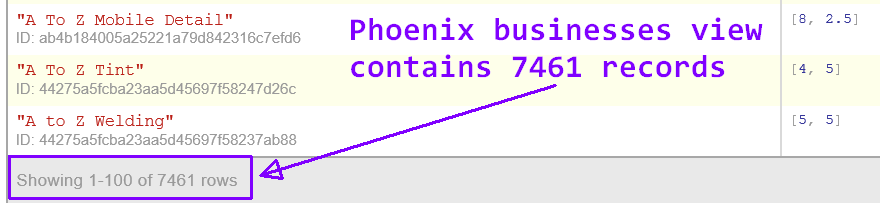
**Update**

To update a document, it is very different than doing an update in MySQL. In MySQL, you tell the data base which record to update and what to update and it just does it. With CouchDB you must retrieve the document you want to update with a GET request, change the part of the document you want updated, and then issue a PUT request to re-write the updated document to the database. In the PUT request, you must specify the current revision number as part of the document or the update will fail. If the update is successful, CouchDB will respond with **ok : true** followed by the document id and a new revision number. In the image below, I updated the **GoodForKids** key from False to True for the document with an id of d9f913332d66beb26da5290bf320a390.

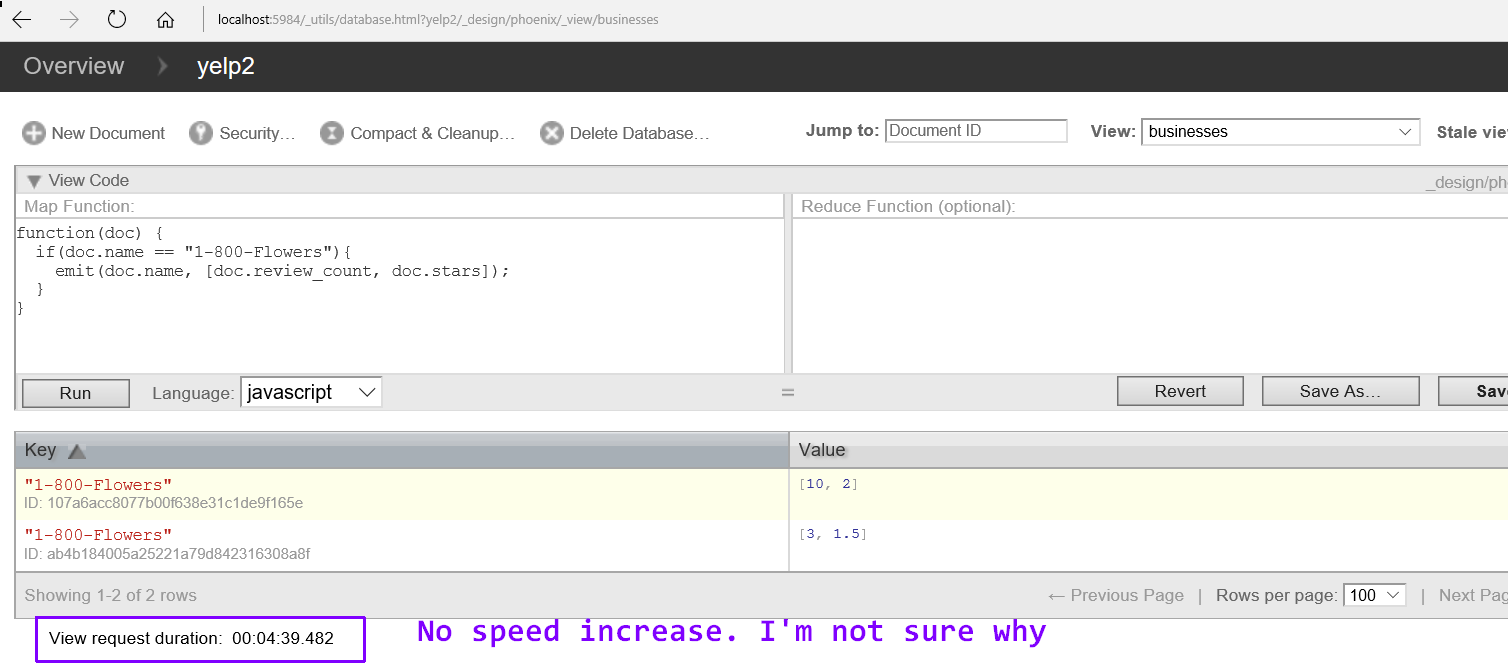


**Optimization**

To see how much faster a query is run against a view versus running over all the documents in the database, I ran the map function below against the Phoenix businesses view. This view only contains 7461 records, so I would expect a query run against this view would be much faster than any query run against the whole database.

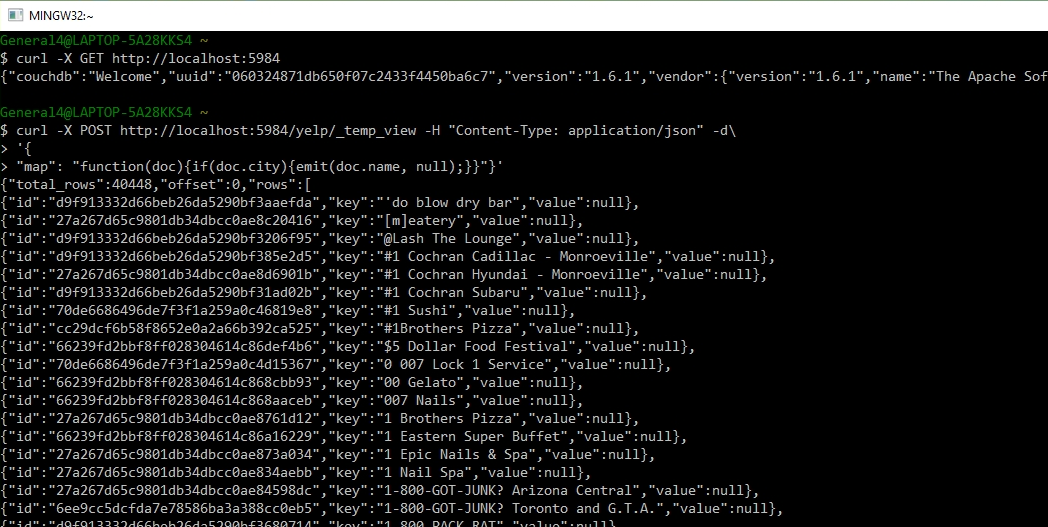


To create the Phoenix businesses view, the query must be run against every record in the database. As noted above in the Query #3 section, it took 4 minutes and 40 seconds to create the view. Since each view in the database is stored in it’s own B-Tree index, I would expect any query run against this view to take much less than this. However, this was not the case. In the query shown in the image below, businesses with the name 1-800-FLOWERS were selected. This query took about the same time as running a query agains the whole database. I am obviously missing something very important. I have no idea why this happened.



**Just a note:**

I could have used the command line more for my queries, but I chose not to because the output was not always well formatted. I did notice though, that the queries ran faster through the command window. The query shown in the image below lists business names. The business documents are the only documents with a key named city, that is why I used that as a filter condition.



**Summary**

This NoSQL assignment turned out to be a challenge. I think the pressure of being loaded down with end of semester assignments weighed on my shoulders and definitely had an impact on this report. It seemed like I stumbled onto one obstacle after another. I didn’t even mention in any of the sections above about how much time I waisted because version 2 of CouchDB would not run on my Windows 10 machine. I tried several times and had no luck. I could get version 2 to run on my older computer that has Windows 7, but not on my main laptop. Therefore, I ended up installing version 1.6.1. This was frustrating since one of the reasons I picked this technology was that it had a Windows installer.

On the bright side, I did gain an appreciation for the NoSQL methodology. A document store seems like a good way to implement a database. Data integrity is really up to the programmer. The database will not save you from loading bad data. As long as it is in well formed JSON, it will accept it. So, for instance, CouchDB will not complain if you have a string named ‘crap’ as a value for a key named ‘price’. In MySQL, in such a situation you would likely have an attribute based constraint specifying that price should be a float. This schema-less database design puts more responsibility on the database programmer to ensure data integrity and consistancy.

I’m not sure that the way I stored my data in the database was the best way. I knew that all the documents needed to be in the same database, but maybe I could have grouped them into several large documents versus many small documents. If this was possible, it might have increased the query times since there would be less file accesses. I think it is pretty cool how CouchDB uses a RESTful API and leverages the JavaScript language. JavaScript is the main scripting language of the internet so this makes a lot of sense to me. I’m not as strong with JavaScript as I wish I was, so I definitely learned a lot with this assignment. I imagine in a production environment, the JavaScript code would be a set of files on a server that are executed when information is needed. Overall, this was a very enlightening assignment.