Gordon Wall

Database Management

Exam 3 – Dr. Brandyberry

Choice 2: Find or conceptualize a hypothetical scenario where a NoSQL database seems to be the right solution.

My scenario involves one in imitation of eBay's platform; one where end users have the ability to search from a near unlimited number of listed products, view said searched-for products, and also post their own items for sale. In this web-based marketplace scenario, complete with mobile application support, users need the ability to successfully search for any product to any n^{th} level of specification, while also being able to customize any 'for sale' posting in the same manner. For this type of scenario, only a NoSQL database solution would suffice in storing all this product information.

A NoSQL database is the right solution for this scenario. Why? Because this scenario requires schema flexibility, scalability, and high performance/functionality; all things a NoSQL database can provide that a traditional SQL-based, relational database cannot.

Schema Flexibility:

The online marketplace problem experiences massive amounts of semi-structured and unstructured data being inputted every second. NoSQL databases provide flexible schemas that do not require rigid structuring and referential integrity while designing the database like SQL databases do. In fact, NoSQL databases are regarded for their 'on-the-fly' ability to alter and restructure schema in a way that can process and store writes to the database in any way they're received. Thus, flexibility in schema structure becomes a must for this scenario when users are posting items for sale; items that have an infinite amount of product specification combinations. The example I provide in my mock application design below shows several products that illustrate this point, each with their own categories and subcategories that store different and unrelated types of data. A classic rows and columns schema that enforces population of the same data across set columns for each item wouldn't work in this setting. For example, a classical guitar listing would contain product detail about the body type and stock strings, where an iPad Pro listing would include different and separate product detail about storage capacity, screen size, and connectivity. It is such, then, that the database would need to be flexible enough to retain this information about individual listings regardless of the data fields present in other stored listings.

Scalability:

A scenario like this also demands scalability. In an instance such as modern eBay, hundreds of millions of listings have been posted, searched for, and/or sold since the marketplace's conception. In this regard, NoSQL databases are perfect for scaling in that they can scale horizontally out by sharding information across distributed hardware, versus SQL databases which utilize vertical scaling. Vertical scaling gets evermore expensive when it's necessary to continually add expensive servers as the business grows. Thus, NoSQL remains the solution for an online marketplace scenario where growth is near exponential and relatively unpredictable.

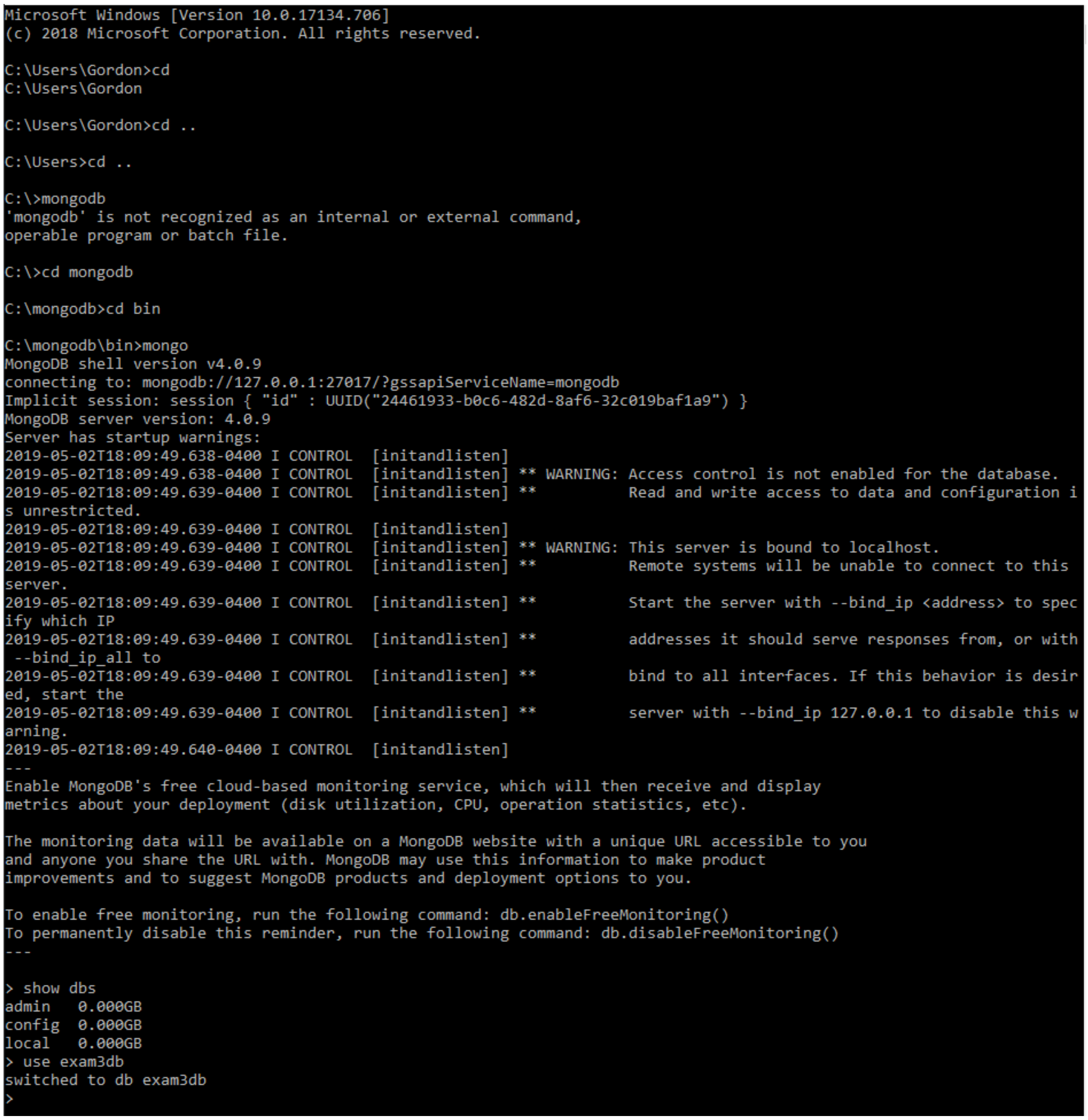
High Performance/Functionality:

Finally, this scenario requires alternative methods of storage; means of storage that deviate relational tables that are linked across foreign keys and contain separated pieces of information regarding individual records to control redundancy. Further, this scenario requires functionality such that its chosen database can interact with a diverse range of APIs. NoSQL databases are perfect for these two remaining stipulations because they effortlessly handle specific storage methods integrated to the applications design (like JSON documents) while also supporting a wide range of functionality in their ability to translate across many different data types. Therefore, NoSQL appears to be the right solution for this mock eBay scenario.

Given the conclusion that this scenario requires a NoSQL database solution, MongoDB presents itself as an optimal means to implement that solution. MongoDB integrates JSON document storage capability that is optimal for the both the storage and retrieval of information about given individual product listings pulled from the mock online marketplace. In contrast with a relational database, this Mongo database (nonrelational) will store all information about an individual product listing with a single JSON document object, which it will then generate a unique object ID for each listing that can be called upon in future searching. Further, this data storage method allows me (the developer) to store unique data fields specific to each listing (and, therefore, each unique object) within those JSON documents. My mockup and screenshots below illustrate several different products being stored. The guitar listing contains fields for product type (instrument), type of instrument (guitar), subtype of instrument (classical), body style (cut-away), strings (nylon), and electronics (built-in). *Different* fields are present and stored for the iPad Pro listing, which contain unique information about brand (Apple), storage capacity (128GB0 and the like. The Mongo database supports this sort of flexibility in design and can accommodate any type of listing inputted in the marketplace and written to the database, shown in the 'Post an item for sale' mockup page. Further, the database stores all this information pertaining to each listing in the form of a JSON document; each field is stored together in an object with a unique ID that can be referenced by the search function of the marketplace during read operations (also shown below). In the mockup, a user can begin searching for items on the ‘Search for an item” page and the application can read from the database in a real-time manner to retrieve the desired information and subsequent product detail contained within each object. Even more so, the application can suggest listings based on the letters the user has begun to type because of its high-performance ability to scan across millions of objects for letter similarity in virtually no time. Finally, when a user has selected an item listing, the application can read from the database all the relevant product information within the referenced object and display it for the user. This is scalable as the business grows and more listings are written to the database.

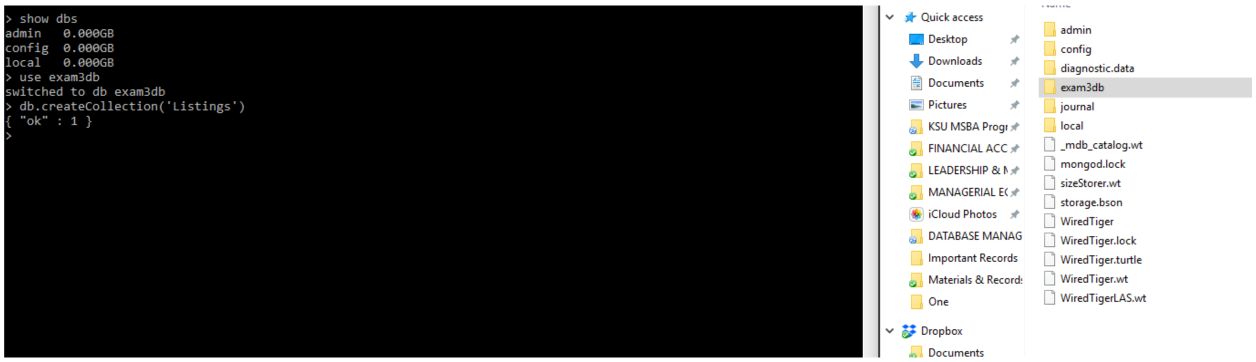
In a relational database, these aforementioned operations would take entirely too long to process and are not suitable for modern applications where real-time feedback is crucial. Similarly, processing a massive amount of semi-structured data, like the data written to the database in new listings, would be outside the capability of an SQL database. Finally, when it comes to a scenario as large as this one, only nonrelational database solutions would be scalable enough to grow with the business without breaking the bank on expensive server additions. In conclusion, a NoSQL database like MongoDB is the perfect solution for this job.

Below are screenshots of MongoDB for support with additional comments (Mockup of web-based marketplace is attached separately as a PDF):



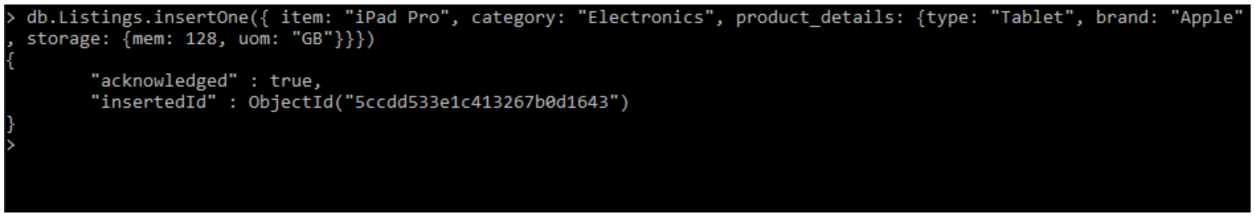
```
Microsoft Windows [Version 10.0.17134.706]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Gordon>cd
C:\Users\Gordon

C:\Users\Gordon>cd ..

C:\Users>cd ..

C:\>mongodb
'mongodb' is not recognized as an internal or external command,
operable program or batch file.

C:\>cd mongodb

C:\mongodb>cd bin

C:\mongodb\bin>mongo
MongoDB shell version v4.0.9
connecting to: mongodb://127.0.0.1:27017/?gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("24461933-b0c6-482d-8af6-32c019baf1a9") }
MongoDB server version: 4.0.9
Server has startup warnings:
2019-05-02T18:09:49.638-0400 I CONTROL  [initandlisten]
2019-05-02T18:09:49.638-0400 I CONTROL  [initandlisten] ** WARNING: Access control is not enabled for the database.
2019-05-02T18:09:49.639-0400 I CONTROL  [initandlisten] **          Read and write access to data and configuration i
s unrestricted.
2019-05-02T18:09:49.639-0400 I CONTROL  [initandlisten]
2019-05-02T18:09:49.639-0400 I CONTROL  [initandlisten] ** WARNING: This server is bound to localhost.
2019-05-02T18:09:49.639-0400 I CONTROL  [initandlisten] **          Remote systems will be unable to connect to this
server.
2019-05-02T18:09:49.639-0400 I CONTROL  [initandlisten] **          Start the server with --bind_ip <address> to spec
ify which IP
2019-05-02T18:09:49.639-0400 I CONTROL  [initandlisten] **          addresses it should serve responses from, or with
 --bind_ip_all to
2019-05-02T18:09:49.639-0400 I CONTROL  [initandlisten] **          bind to all interfaces. If this behavior is desir
ed, start the
2019-05-02T18:09:49.639-0400 I CONTROL  [initandlisten] **          server with --bind_ip 127.0.0.1 to disable this w
arning.
2019-05-02T18:09:49.640-0400 I CONTROL  [initandlisten]
---
Enable MongoDB's free cloud-based monitoring service, which will then receive and display
metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible to you
and anyone you share the URL with. MongoDB may use this information to make product
improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---

> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
> use exam3db
switched to db exam3db
>
```

The picture above shows successful installation of MongoDB on User/Gordon’s PC, set up to run as a service, and display of default dbs (<show dbs>) with new use and creation of *exam3db* for this project.



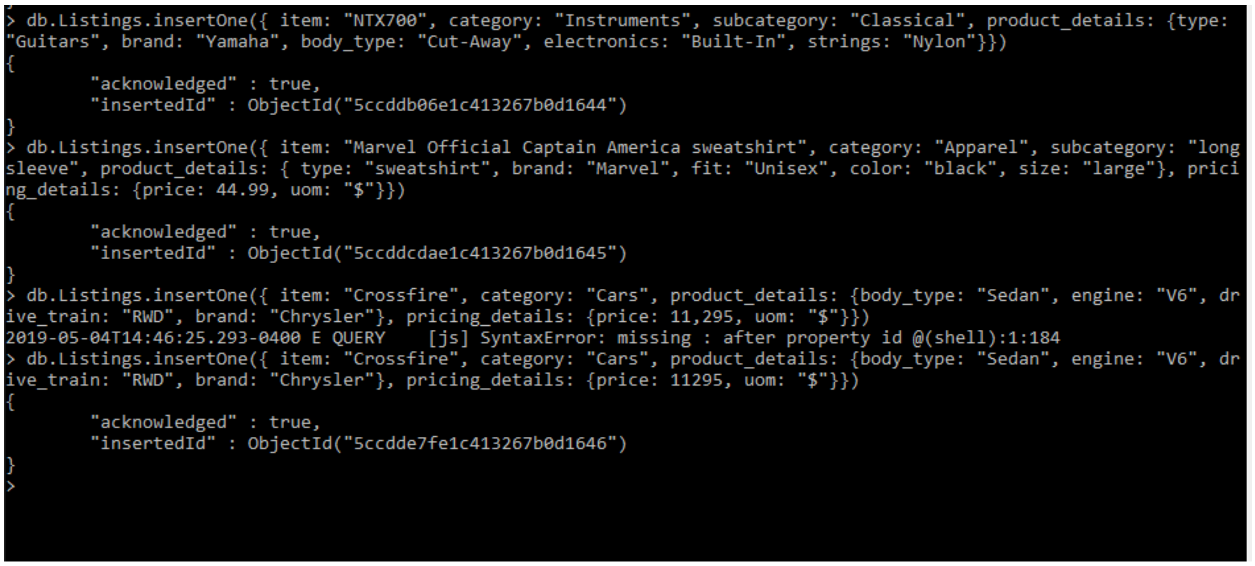
```
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
> use exam3db
switched to db exam3db
> db.createCollection('Listings')
{ "ok" : 1 }
>
```

This second picture (above) shows successful creation of *exam3db* along with the creation of a collection, *Listings*. Our mock marketplace users will create listings for sale, and they will write to this collection to be stored and referenced in a variety of ways later.



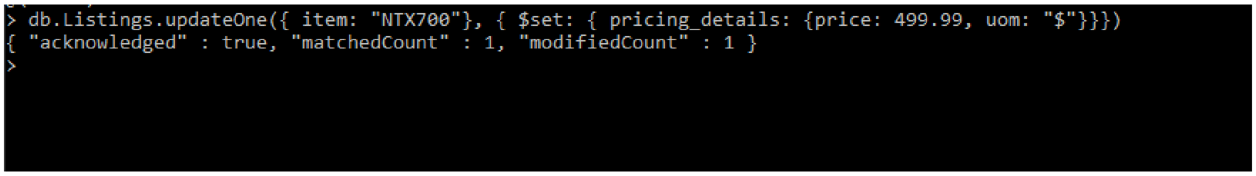
```
> db.Listings.insertOne({ item: "iPad Pro", category: "Electronics", product_details: {type: "Tablet", brand: "Apple"
, storage: {mem: 128, uom: "GB"}}})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5ccdd533e1c413267b0d1643")
}
>
```

This third picture (above) shows the write of our first user's listing for an iPad Pro. The first screen mock up included with the PDF shows how a user can build a listing on the 'List an Item for Sale' page without adhering to strictly defined schemas (like in a relational db). A user can fill out any necessary amount of detail to be included with their listing so other users may browse, shop, and buy based on that detail. The MongoDB stores these attributes as fields in a JSON document pertaining to one listing with one unique object ID.



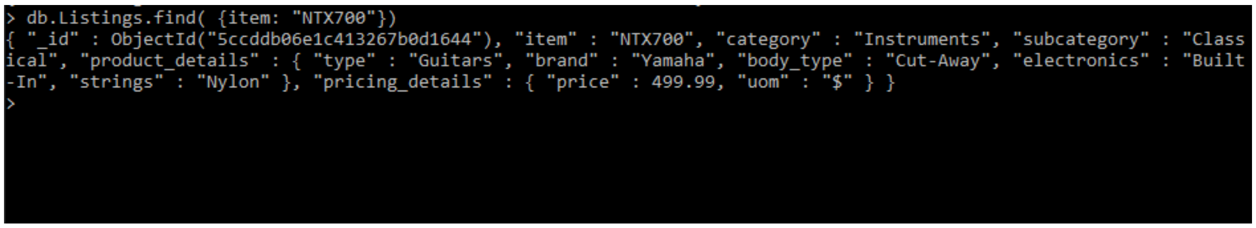
```
> db.Listings.insertOne({ item: "NTX700", category: "Instruments", subcategory: "Classical", product_details: {type:
"Guitars", brand: "Yamaha", body_type: "Cut-Away", electronics: "Built-In", strings: "Nylon"}})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5ccddb06e1c413267b0d1644")
}
> db.Listings.insertOne({ item: "Marvel Official Captain America sweatshirt", category: "Apparel", subcategory: "long
sleeve", product_details: { type: "sweatshirt", brand: "Marvel", fit: "Unisex", color: "black", size: "large"}, prici
ng_details: {price: 44.99, uom: "$"}})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5ccddcdae1c413267b0d1645")
}
> db.Listings.insertOne({ item: "Crossfire", category: "Cars", product_details: {body_type: "Sedan", engine: "V6", dr
ive_train: "RWD", brand: "Chrysler"}, pricing_details: {price: 11,295, uom: "$"}})
2019-05-04T14:46:25.293-0400 E QUERY    [js] SyntaxError: missing : after property id @(shell):1:184
> db.Listings.insertOne({ item: "Crossfire", category: "Cars", product_details: {body_type: "Sedan", engine: "V6", dr
ive_train: "RWD", brand: "Chrysler"}, pricing_details: {price: 11295, uom: "$"}})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5ccdde7fe1c413267b0d1646")
}
>
```

Picture four (above) shows the creation and insertion of three more listings to the MongoDB; a Yamaha classical guitar, a Chrysler Crossfire car, and a Captain America sweatshirt. The db stores these with their own JSON documents and unique object IDs as well.



```
> db.Listings.updateOne({ item: "NTX700"}, { $set: { pricing_details: {price: 499.99, uom: "$"}}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
>
```

The user can change/add to their listing down the road (show in picture 5 above). Here, they go back to their posted listings and set the price to $499.99. A user may also delete their listing if they change their mind about selling. There is no screenshot for this, however, the command the database would run when the front-end user deleted this listing would be <db.Listings.deleteOne({ ObjectID: 5ccdde7fe1c413267b0d1644 })>.



```
> db.Listings.find( {item: "NTX700"})
{ "_id" : ObjectId("5ccddb06e1c413267b0d1644"), "item" : "NTX700", "category" : "Instruments", "subcategory" : "Class
ical", "product_details" : { "type" : "Guitars", "brand" : "Yamaha", "body_type" : "Cut-Away", "electronics" : "Built
-In", "strings" : "Nylon" }, "pricing_details" : { "price" : 499.99, "uom" : "$" } }
>
```

The final picture (above) references the second screen in the mockup PDF, the search function. Without designing a full application with front and back end support, the web browser search function could search across all objects (JSON documents) and locate listings that matched the search criteria. It could also begin doing so as the user typed, like Google, where the results box could begin auto-filling suggested results. In the mockup, the user begins typing the letter ‘C’ and our three stored listings with the letter C in the product details appear as search recommendations. These are all high performance/functionality benefits of using a NoSQL database.

Finally, the marketplace could display all information about a listing once a user selected an item to view and/or buy. The third mockup screen shows this function. The browser reads from our NoSQL database, references the selected object, and displays all the attributes contained within. Here, the user views the Yamaha NTX700 Classical Guitar.

[All mock up screens were done in Adobe to supplement the theoretical design of this scenario and web-based marketplace. They are not intended to be complete or usable but, rather, are tools for illustration.]