**COMP.3100 Database II Spring 2022**

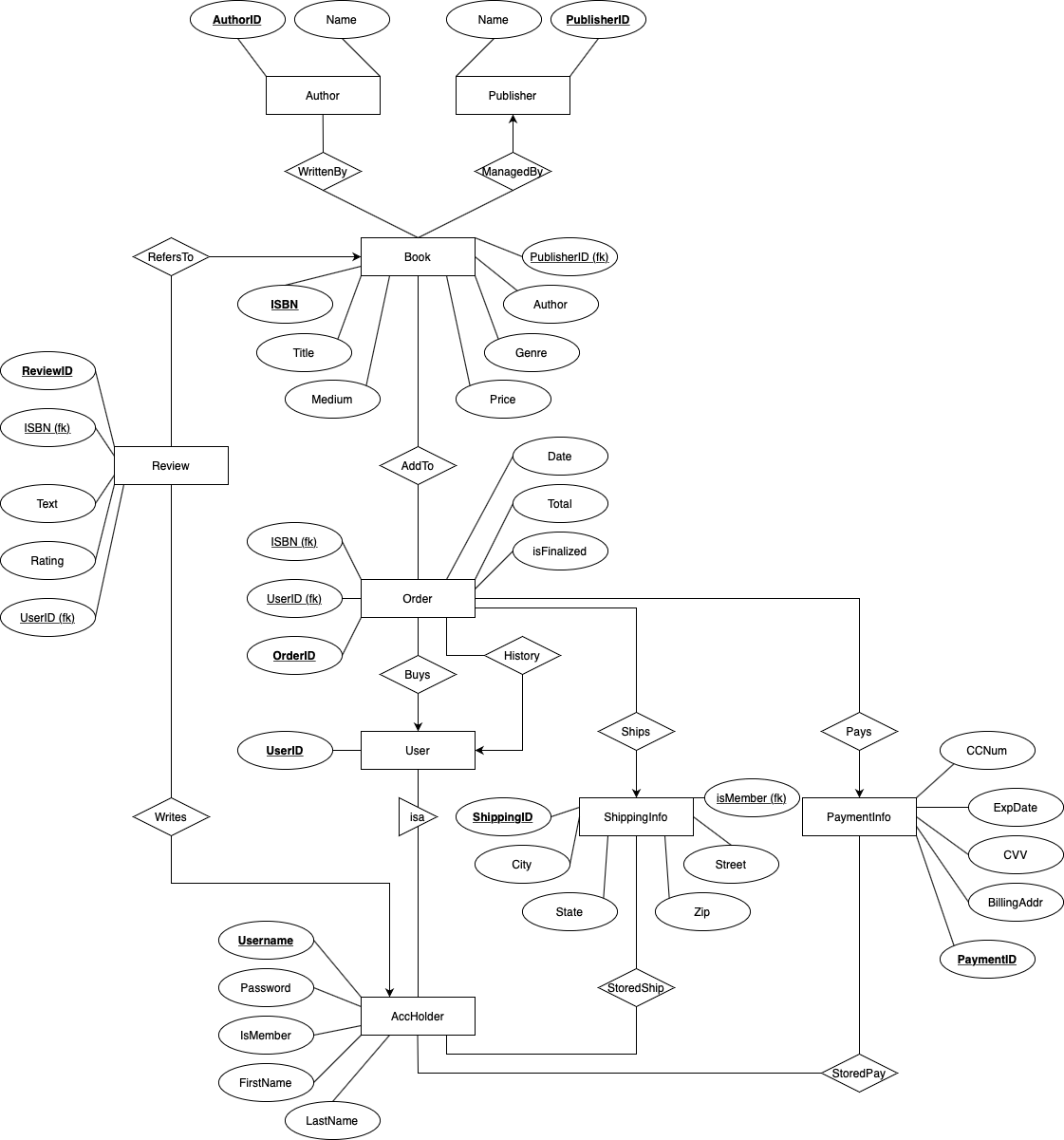
**Term Paper**

Author: Kyle Savoie

**I. Approach**

In this class, my group and I were tasked with creating a functional online bookstore comparable to existing online stores like Amazon. To do so we used many technologies and methodologies separated into three phases. In Phase 1 we applied what we learned about ER Diagrams in the first section of this project sequence to create a schematic representing what our actual SQL database would look like. In Phase 2 we created our database using SQL, hosted it using PHPMyAdmin, and created a functional website with HTML and PHP. In Phase 3 we followed a similar process to create an Android application in AndroidStudio using Java, PHP, and XML that included a few of the functionalities from our aforementioned website.

**Phase 1:**

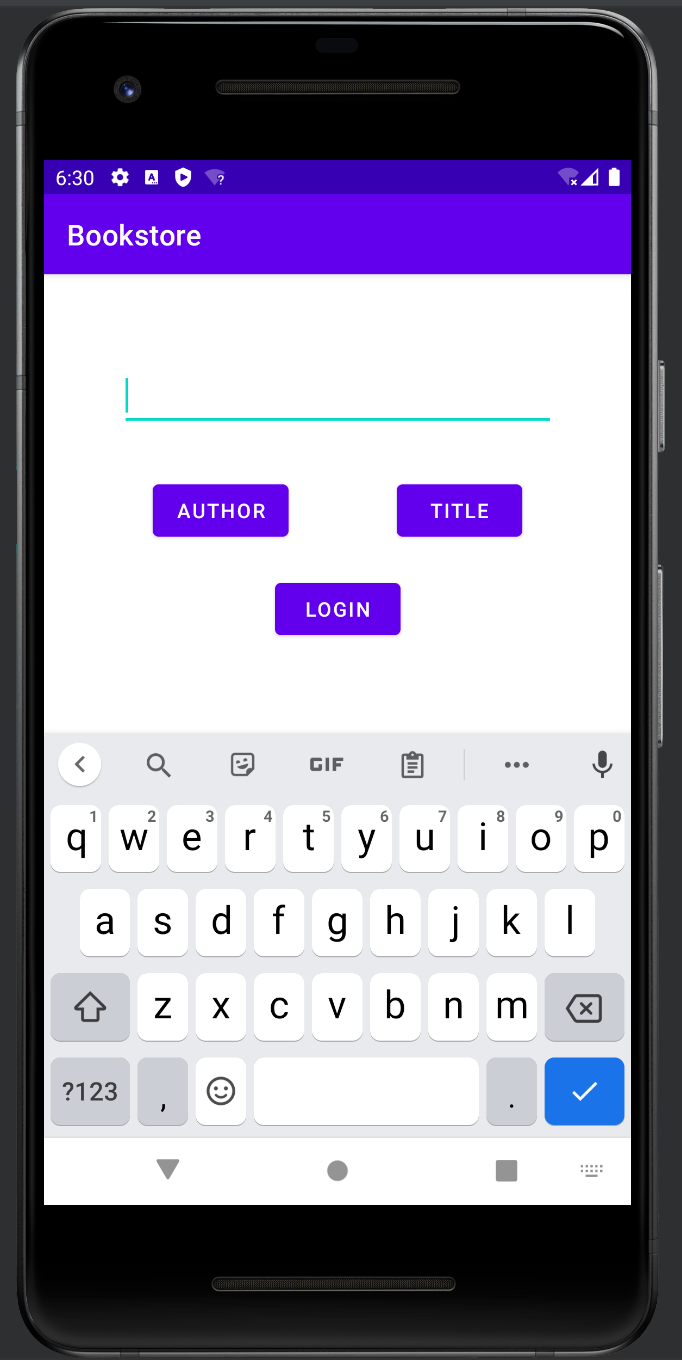


Shown above is the final ER Diagram from Phase 1 of our online bookstore project. Though the process of creating an ER Diagram is conceptual, it is a process that requires frequent adjustment as one learns more about the problem they are trying to solve. In our case, we first created an ER Diagram to our best abilities which would make room for all of the necessary entities, relations, and attributes in a thoughtful way, but as we neared the end of our list, we noticed that our diagram was far from optimized and that our “logical” approach did not take into account the actual process of converting the diagram to a functional database. It became apparent that there lies a difference between creating a strictly “organized” ER Diagram and creating one that is both organized and functional. Because of this, we decided to recreate our diagram with a new approach, keeping in mind the implementation step more so than before. We prioritized intelligent relations, while also understanding that it may not be conducive to the easy creation of an SQL database if we minimized the number of relations or connections between entities. With this in mind, we wrote our SQL queries before the creation of the ER Digram which allowed us to better understand how to create a diagram that wouldn’t make our Phase 2 much more arduous. This led us to the creation of the ER Diagram above.

**Phase 2:**



The above image shows the landing page for our Online Bookstore website. Phase 2 was a multi-part phase. The first task was to convert our ER Diagram from Phase 1 into a SQL database. This was a simple process because we only needed to convert the visual layout of the diagram into SQL “create table” statements. The only added complexity was adding the foreign key checks into the “create table” statements, but these were also represented in the diagram. Lastly, we added “insert” statements to populate the tables so that we could test the functionality of our website. The next step was to host the database with XAMPP and Apache. We configured and launched a MySQL Database and an Apache Web Server. The next and final step of Phase 2 was by far the most complex of the project. For each page and functionality of the website, we had to create a PHP file with included HTML if we were designing a displayed page. This came with quite the learning curve for me since I had never used PHP before and had limited experience with web development. We were able to quickly understand HTML and implement the “visual shell” of our website, that being all of our required pages displaying correctly, but lacking any connectivity or functionality. After working more with PHP, we were able to understand the process behind transferring data with \_SESSION variables, redirecting pages with the header() function, etc. Personally, I primarily dealt with the account side of our online bookstore website. I handled the account and publisher registration and login, the elevation of account holders to members, the corresponding landing pages for each account type, and the payment and shipping update pages. I also designed the index.php landing page and the header to be used on all pages. I was able to conveniently group these pages together because much of the functionality that exists between these pages does so in a closed system, so each group member could work on their own corresponding section of the website, then link them together towards the end of production. registerPublisher.php and registerAccount.php were logically the first pages I began to work on once index.php was designed. These pages require basic information from the user to populate the fields necessary to run an INSERT query into the appropriate table, thus creating the desired account. Most of this was quite straightforward after some practice and these pages acted as training grounds for me to design a template to use for many other pages in the site. One difficulty was figuring out how to store, and then display errors. I decided to create an array of errors that would print out at the bottom of the page if something was wrong with the user’s input. loginPublisher.php and loginAccount.php both followed similar design patterns. Both pages used PHP to pull the username and password from the text boxes, save them into variables, then query if the account exists. If it did, the pages would set \_SESSION variables for the “userMode” which identifies the account type, and “id” which identifies the ID of the logged in account. Simple HTML displayed the page that the user interacted with. Unlike these two pages, continueGuest.php did not include any HTML. It simply set the \_SESSION variables and redirected the user to the main bookstore page with limited permissions. The elevation of an account holder to a member was a collaborative effort and we decided on a method that used our current infrastructure. Rather than designing a new item type, we created a “book” with data that would not be generated naturally. When a user checked out with the “Become Member” item in cart, the account would be updated to member status and the shipping for the current, and all future orders would become free of charge. publisherLanding.php, accountLanding.php, and adminLanding.php served as the home page for each of the corresponding account types. These pages were only accessible to users that held the correct “userMode” \_SESSION variable and included all of the correctly permissioned functionalities of each account type. Similar to the method we used to elevate account holders to members, we created an admin account with data not possible for accounts created through the website. Therefore, if a user logged in with admin credentials, the website would know, and properly set the “userMode”. adminLanding.php would then only be accessible to the admin account. continueGuest.php would simply redirect to the online bookstore’s main page, and had no account type specific permissions and therefore did not have a corresponding landing page. updateShipping.php was essentially a visual “UPDATE” query. Each box corresponded to a data point in the stored shipping. When a user entered new information, the website would use basic input sanitization, then run an “UPDATE” query to change the stored shipping information. I also designed updatePayment.php to do the same for stored payment information, but I was unable to fix some small problems and ultimately removed it from the final submission. Lastly, I created header.php which included the CSS style for the pages to inherit as well as the header or banner that would be displayed at the top of each page, customized with the page’s title. This helped to make the website seem a bit more sophisticated as opposed to black and white, and made it a bit more readable and user friendly. This phase was a success and I was very happy with what my group members and I created.

**Phase 3:**

The image to the left shows the landing page of our Android app created in AndroidStudio using a combination of Java and PHP. Though this phase only tasked us with implementing search, login, and order history functionality, it was quite difficult due to the amount of research into AndroidStudio and Java required before making any progress. Personally, I had much trouble getting the Android emulator to work and the sample app to connect to my SQL database due to compatibility issues. Once this was finished, my group members and I began to research the Volley library that was used throughout the sample app as well as brushing up on general Java. Personally I designed the landing page and did a large amount of research into displaying an JSONArray array of JSON objects onto a page. The landing page was quite easy thanks to AdnroidStudio’s drag and drop feature. I had some trouble getting everything to display correctly, but after some edits to the XML code I was able to get everything working perfectly. As for the JSONArray issue, this by far caused my group members and I the most trouble. The problem was that in the sample app, only one movie from the search results were displayed, but in our app, we needed to display every book that matched the search result as well as display every order in an account holder’s order history. The sample app passed raw text into a JSONObject, but our app needed to pass an array of JSONObjects in a JSONArray, then display each JSONObject in a separate card on the same page. I personally spent quite some time searching for a solution for this while my other group members designed the remaining pages and PHP files. Luckily, I was able to find multiple resources showing solutions to vaguely similar problems, allowing us to get a better understanding of the process than the Volley documentation was able to show us. Through a combination of RecyclerViews and TextViews we were able to correctly display the information using our desired method of a JSONArray of JSONObjects. The final app successfully fulfilled all of the required functionality.

In each phase, each of the group’s members were able to use their strengths to best serve the group as a whole. I believe that we worked exceptionally well together, and divided the work evenly. As a result, I believe that each group member contributed equally (33.3%) to the assignment’s completion. In Phase 1, we worked very closely together on the ER Diagram. In Phase 2, we split the work into separate sections and worked independently, linking all of our work together towards the end of the phase. In Phase 3, we divided the work into sections similarly to Phase 2, but worked closely together to solve the problem of JSON parsing.

**II. Amazon’s Dynamo**

Amazon's DynamoDB is a solution to their extreme database scalability and data availability requirements that a relational database management system could not efficiently support. DynamoDB is a key-value store that uses consistent hashing and object versioning in a primary-key only interface to trim the unnecessary features and compromises of a RDBMS, creating an efficient database solution for applications which prioritize database scalability and data availability over consistency. Because of this specialization that DynamoDB focuses on, its application becomes more limited, while it excels in the applications it was designed for. Though there exist many advantages and disadvantages to DynamoDB, this analysis will focus specifically on three strengths and two weaknesses of this database implementation: Incremental Scalability, Symmetry and Decentralization, customizable database functionality, a simple Query Model, and the possibility of versioning errors.

Database scalability is extremely important to Amazon for a multitude of reasons. In relation to their DynamoDB implementation, which deals with, for example, their shopping car service which may increase in load multiplicatively during times of high demand during the holiday season, having the ability to scale these services in response to an increase in demand is imperative to running a profitable and functional service. To address this, DynamoDB allows for the addition or deletion of nodes into their distributed system of hosts without impacting the entire system. To partition new nodes, DynamoDB uses a modified version of consistent hashing. Traditional consistent hashing utilizes physical nodes. The hash output function of these nodes can be visualized like a ring where each node is randomly assigned a position on the ring. Every data item is then assigned to a node, making each node responsible for the area between itself and the node prior. The strength in this implementation is that nodes can be added or deleted while only affecting its neighboring nodes, but more modifications needed to be applied to properly optimize for DynamoDB's intended use cases. The issues with standard consistent hashing are as follows: nodes' random assignment to position on the ring leads to in-consistent load distribution, and the algorithm does not take into account that each node may possess different levels of hardware capabilities, thus ignoring heterogeneity. DynamoDB solves these issues by implementing virtual nodes. Virtual nodes appear as a single node, but may be responsible for the coverage of multiple virtual nodes. Removing a virtual node spreads that node's responsibility evenly among the remaining virtual nodes. Adding a node prompts said node to accept responsibility from every existing node at an equivalent amount. A more powerful physical node may accept responsibility over more virtual nodes than a less powerful physical node, thus accounting for heterogeneity requirements. Because of these changes, DynamoDB's virtual node consistent hashing method becomes even more scalable than traditional consistent hashing, allowing DynamoDB to safely be identified as a highly scalable database solution.

If scalability allows the network to grow or shrink alongside the necessary level of demand thus increasing the network's functional power, decentralization allows the network to be more resilient in the face of server outages thus increasing the network's functional availability. As opposed to having a DynamoDB hosted on one central server, the database is supported by many servers running identical nodes. Because of this, a server outage that would result in a complete service outage in a centralized system would affect a decentralized system to a much lesser extent, simply decreasing the load that the entire decentralized system may handle. An important factor of this decentralization lies in the symmetry of the virtual nodes hosted on each server, or physical node. Symmetry guarantees that each node holds the same responsibilities and data to ensure that no one node is imperative to the functionality of the system as a whole. To further bolster network and data availability, the distributed nature of the DynamoDB file system allows read and write operations even during network partitions and is able to elegantly handle the resulting conflicts. Due to its specific application inside of Amazon's suite, data integrity and security is handled outside of the database. Due to these qualities of DynamoDB, if applied to the correct task, the database will hold an increased level of availability due to its resiliency to server failures. It will also experience an increase in data security because its symmetrical properties will guarantee that data is mirrored across every hosted node. Data replication is an important factor in data availability and as a result of decentralization and symmetry is automatic in DynamoDB.

Incremental scalability, symmetry, and decentralization are fundamental design choices in DynamoDB, but like many complex architectures, there exists a level of flexibility inside the highly specialized environment to further adjust the technology to better serve its intended purpose. DynamoDB allows users of its client application to tune its performance to hold greater weight over certain performance metrics, further defining its specialization. In DynamoDB there are three values a user may adjust, N, R, and W which affect the database's performance, availability, and durability. N represents the number of hosts or nodes, R represents the minimum number of nodes that must participate in a successful write operation, and W represents the minimum number of nodes that must participate in a successful write operation. Adjusting the W and R values will change the availability of objects, as well as durability and consistency. This level of adjustment allows for customers to fine tune their instance of DynamoDB to their specific use case. This form of adjustment is not available in traditional RDBMS.

DynamoDB specializes in its scalability and data availability, but does so knowingly at the detriment of other metrics. While the database design itself is catered towards certain tasks, DynamoDB lacks in certain areas compared to a standard RDBMS. One of the most glaring functionalities that DynamoDB lacks due to fundamental design choices is the ability to conduct anything more complex than the most basic queries. DynamoDB uses key-value storage. Data items are uniquely identified by a key and state is stored as binary objects which are also identified by unique keys. Because the intended purpose of DynamoDB is not to handle complex queries and instead be applied only in environments where quick, reliable, and simple queries, the database cannot handle relational schema and queries. Instead, DynamoDB can only handle get() and put() queries. This is not a weakness in perfect applications, but in scenarios where more is needed from DynamoDB, the database is unable to adjust to the customer's need, and another solution must be implemented. This may limit DynamoDB's application in the industry. If a customer needs more than primary-key queries, for example, to query a multi-attribute-key table, a table including foreign-key constraints, or a table spanning multiple data items, they will be forced to look elsewhere.

Finally, it is important to touch on the negative side effects of DynamoDB's always readable and writable nature. Once again, in the specific application DynamoDB serves, this allows for strengths in more preferred areas, but in applications that require accuracy and consistency, DynamoDB has an immense issue. That issue is the idea of multiple versions of a data instance existing at once. A put() call may be returned to the caller before the update from the put() has been applied to all replicas of the data instance. Because of this, a get() call on an out of date server may return different results than a get() call on an updated server, despite the customer making the calls from the same hosted service. Furthermore, if outages occur, the updated information may never reach all of the servers, and some servers may hold incorrect information for an extended period of time. DynamoDB has implemented a partial solution to this problem with the help of vector clock information, but states that it may lead to inaccurate data, though it has not been experienced in a production environment, This would be extremely detrimental to a application where accuracy is of utmost importance, for example, a hospital database that holds records of medicine administered to elderly patients. If multiple versions of these records exist and the wrong record is propagated to other servers, an issue would arise.

DynamoDB is a creative solution to the problem of a highly available and scalable database which can make fast and simple queries for businesses operating under extreme load. It has many strengths as a result of its deviances from the traditional RDBMS method, but in turn makes trade-offs that push DynamoDB into a more specialized use case than what is sought to improve upon. It is my belief that more specialized solutions yield better results, and that solutions developed for specialized problems have no place in applications outside of its areas of expertise. Because of this, it is difficult to say that DynamoDB has weaknesses, so much as it has applications where it would perform much weaker than a standard RDBMS. Despite this, DynamoDB successfully solves the problem it set out to fix, and will most likely continue to find industry adoption as time continues.