Polyglot Persistent Data Service

Preston Knepper, Dalton Rogers, and Kevin McCall

CS 496: Capstone II

Western Carolina University

May 10, 2025

Introduction

Databases are central to nearly every modern application. However, displaying large amounts of data is a complex problem. How will the data be stored? Where will it be stored? How can we access the data? Who will have access to the data? What does the data represent? These are important questions to ask when planning an application that requires a database.

Until recently, relational databases built on SQL dominated the database industry. The importance of these systems is illustrated by Wade & Chamberlin: “Relational database systems are among the world's most successful software products and are probably the aspect of the computer revolution that has had the most direct effect on the everyday lives of ordinary people” (Wade & Chamberlin, 2012). Relational databases are strong, easy to understand, and simple to query, with an extremely diverse set of applications.

However, with evolving technological needs, non-relational databases (also called NoSQL databases) grew in popularity. These databases grew particularly fast due to the growing need for scalability, elasticity, and flexibility that traditional relational databases could not provide (Kraska & Trushkowsky, 2013). These NoSQL databases were often designed with a specific use case in mind. Using NoSQL, developers gained the ability to tailor their databases to the unique needs of their applications. As a result, many different kinds of databases have emerged, some of which will be examined in greater detail later in the report.

This project was split into two semesters of development. In the first semester, the project focused on creating a functional polyglot persistent data service. This system consisted of three open-source database systems. Each of these systems is responsible for a specific subset of tasks within the overall polyglot service. These systems were integrated over time, one at a time. In this second semester, we implemented a fourth database system and deployed a distributed

polyglot service through sharding. This allowed the polyglot service to work more efficiently by utilizing multiple machines, which is especially important as the dataset grows large. We were also able to utilize the recently added database in a caching algorithm. It keeps the information the user has recently viewed or, will likely view in the future, closer to them.

Project Timeline

The initial objective of the first semester of this project was to get each of the databases integrated into a singular system. Once each of the databases were integrated, our focus was to shift to creating a REST API. This API would then be used to create an interactive frontend for users. With Hurricane Helene affecting our area along with our school schedule, we were forced to modify our plans. Our planning went from having four databases, to three, and our frontend was unable to be fully completed because of the time constraint. We were able to successfully incorporate PostgreSQL, MongoDB, and Neo4j over the course of the first three sprints. After establishing these databases, our REST API was created with appropriate GET, POST, and PUT requests to be used in various ways by the frontend. For the final half sprint, we spent time putting together a frontend that had limited functionality but provided an outline for the frontend we have at the end of the second semester.

In the second semester, we focused on two primary objectives. One was to make significant improvements on the frontend from the previous semester. The other was to implement a custom sharding algorithm. Kevin joined our team at the beginning of the semester, so we split up our work differently than the previous semester. At the beginning, the plan was to break up the work. One person would work on the frontend, while the other two people would

work on the sharding algorithm. We hoped that we could finish the sharding algorithm early, and have the rest of the semester to do frontend work and testing.

The first two sprints went this way. Redis, our choice for implementing the sharding algorithm, was set up. We created a script in rust that would set up docker containers for our three other databases. We also set up authentication and the tables for PostgreSQL. On the frontend, Tailwind CSS was set up to make styling the site easier. Using Tailwind, a style was finalized and modular components were set up that would lay the groundwork for the rest of the semester. On the backend, the sharding algorithm was completed, but not tested. We had discovered that our sharding algorithm would not work with PostgreSQL and Neo4j, discussed later. A caching algorithm was also completed to speed up GET requests on the API.

The following two sprints we went all-in on our frontend: implementing the visuals for representing our databases, adding pages to create data, and overall completing the frontend in time for the poster session. After the poster session we began working on the sharding algorithm again, only to realize that we had actually completed it during the second sprint. We only needed to refactor some recent code that was made without the sharding algorithm in mind. With that, our project was essentially complete. We migrated the rust script to a Docker Compose file, and worked on building unit tests.

We worked under the following schedule during this semester:

Schedule for Spring 2025		
Date	**Goal**	**Result**
FR, 24 Jan 2025	Have a draft of the proposal nearly finished.	A finished initial proposal draft.
FR, 31 JAN 2025	Have a final draft of the proposal finished.	A completed final draft of the proposal.
FR, 14 Feb 2025	Sprint 1: Focus on making the frontend beautiful. Research, design, and begin to introduce sharding on the backend. Add Redis to the server.	A website with a set theme and style guide. An actionable plan for sharding will be developed.
FR, 7 Mar 2025	Sprint 2: Implement a plan for sharding on the backend. Implement caching. Improve frontend homepage.	The basis for sharding is established on the backend. The website homepage has basic functionality that will carry over to other pages.
FR, 21 Mar 2025	Sprint 3: Polish the sharding implementation. Implement PostgreSQL and MongoDB visuals on the frontend. Turn	The draft poster will be turned in to be revised on March 21st. Visual representations of 2 of

	in the draft poster.	the 4 databases.
FR, 4 Apr 2025	Sprint 4: Implement Neo4j visuals on the frontend. Clean up frontend. Added recommendation cards to product page.	Poster will be turned in on April 4th. Neo4j visualized on frontend and prepped for poster session.
M, 14 Apr 2025	Make final preparations for the presentation and present at the poster session.	The poster session presentation will be complete.
FR, 2 May 2025	Sprint 5: Add Redis visualization to frontend. Complete sharding algorithm. Finialized frontend. Write unit tests using Jest.	The code will be polished and tested to be ready for the final demonstration.
Exam Week	Polish up the project, adding any necessary details. Prepare for the presentation.	Practice presentation and present.
Exam Week	Create the final report.	Turn in the presentation slides and the final report.

Design and Implementation

At the beginning of the project, we listed out several technologies and ideas that we were going to include in our design but decided against as the first semester went on. The first change

was not building our project to take in databases but to instead model an e-commerce application where we create our own mock data. Next, we decided the use of Kubernetes was not needed. Instead the application was hosted on a Raspberry Pi, using Docker to store the databases in containers.

In part to these changes, our overall project design shifted to model an e-commerce application built on a polyglot system. This application consists of users, products, transactions, recommendations, reviews, and ratings. Let us examine the specific databases, and technologies we used to accomplish this goal.

PostgreSQL, as the relational database, offers several advantages as it is extremely versatile and has the capacity to support multiple data types. This support allows both structured and semi-structured data to be managed with grace. Additionally, PostgreSQL ensures data integrity and consistency. This makes it an obvious choice for situations where data needs to be stored and accessed reliably, which will be the case when working with critical data. PostgreSQL is used to store persistent data that a company may want to hold onto, which for us was data about users, products, and transactions. Many companies keep logs on all of the previous users, previous products, and previous transactions. We wish to represent this in our model.

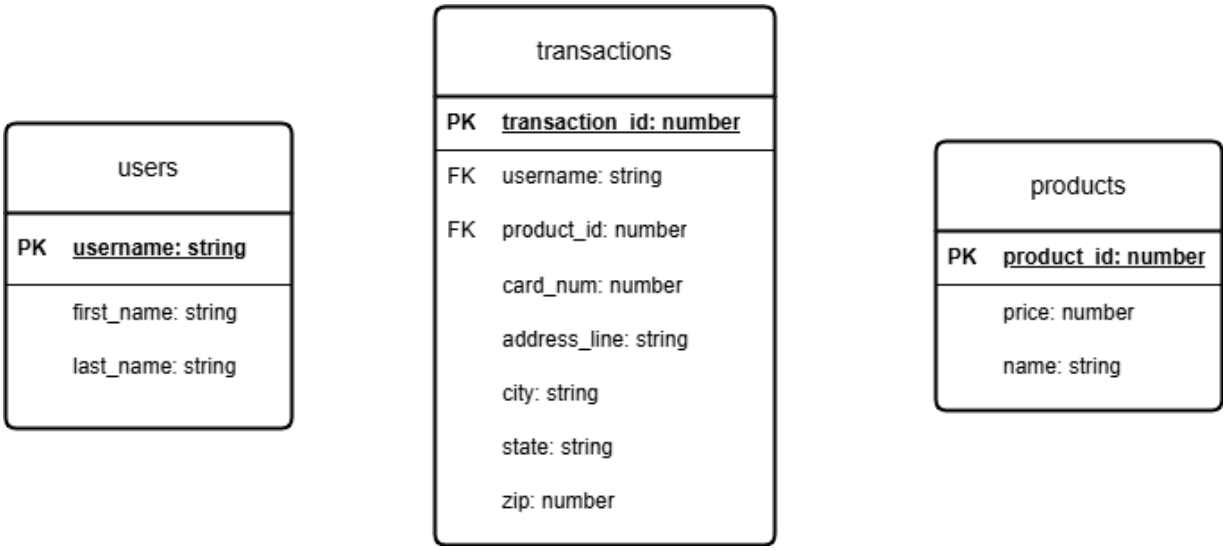

Figure 1: PostgreSQL schema for persistent users, transactions, and products.

MongoDB, a document database, specializes in managing either unstructured or semi-structured data. This database system is designed to store JSON-like documents, which allows objects with nested objects, and there is no enforced schema. As a result, documents may contain unique fields or types that can not be found in other documents. MongoDB is a versatile, easy to use, scalable database. Characteristics such as these were important throughout the course of the project. As a result, MongoDB was used to store our temporary data about current users and products. Many times users and products have varying fields, such as when one user has not purchased anything and another user has purchased many products. MongoDB's flexibility is showcased in situations like this, where the same data type has varying fields, and the way these situations were handled was important in our program.

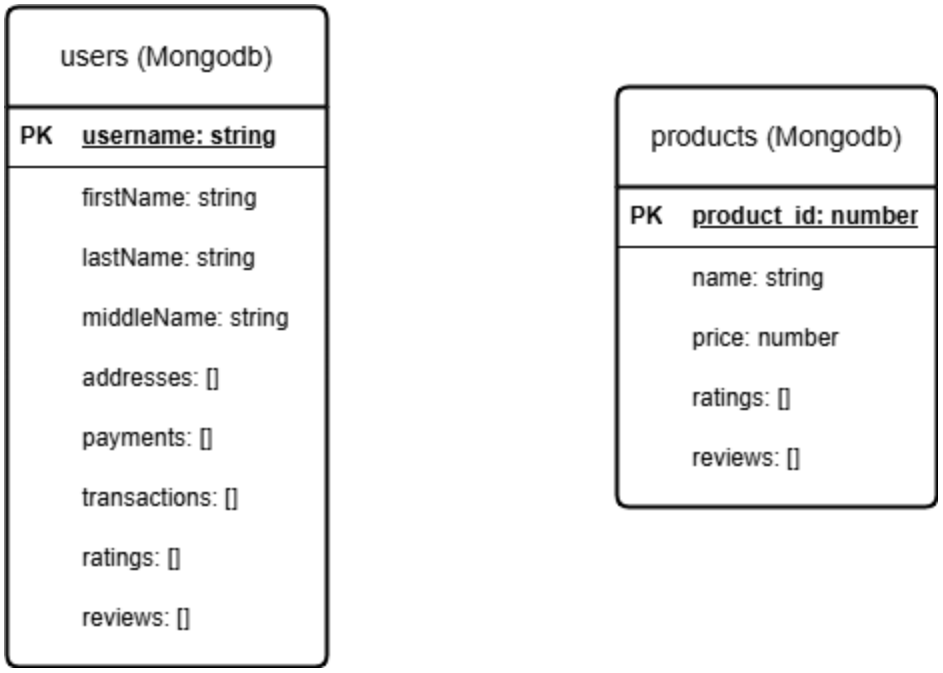

Figure 2: MongoDB schema for current users and products.

Neo4j, a graph database, was crucial when managing and querying data with complex relationships. One of the strengths of this database is its graph-based structure, because of this it quickly draws connections between various objects. This offers some interesting possibilities as we began to implement this into our project, such as recommendation engines. We included this in our project, users can be recommended products based on what other users that bought the same product have purchased. This is similar to what real e-commerce apps do, we have all experienced getting suggestions about what else to buy based on what others have bought. Implementing this capability provides valuable insights that might be missed with other database types. Additionally, Neo4j's ability to support billions of nodes and relationships ensures that it can handle extensive and intricate data sets efficiently, making it a powerful component of our polyglot system.

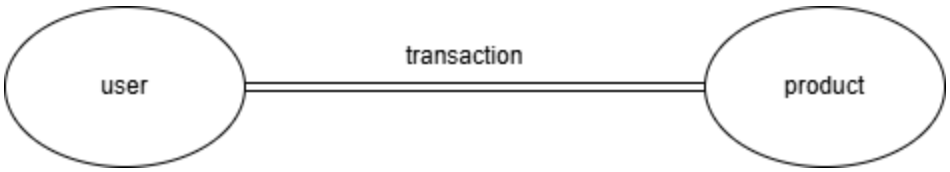

Figure 3: Neo4J schema for showing the relationship (transactions) between nodes (users and products).

Redis is a key-value database. It is extremely fast because it is a hash table stored entirely in random access memory. It only has one table, but we changed our key-naming convention for each type of data we wished to represent. We use Redis for two main purposes: sharding data and caching requests. Additionally, Redis was also useful for synchronizing a counter for our user ids.

The sharding algorithm we implemented with Redis is known as a key-value shard, also called a hashed shard. We lookup a product/user/transaction using an id, and Redis will give us the ip address of the shard containing that specific piece of data.

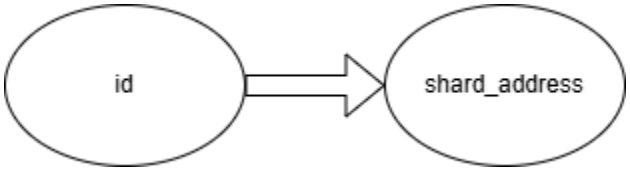

Figure 4: Redis sharding schema for showing which shard a point of data is stored on.

For now, we only implemented our sharding algorithm on MongoDB data. We could not implement it on PostgreSQL or Neo4j due to the fact that entries in both databases rely on foreign keys to other items. Since we cannot access values in entirely different instances of those databases, we are unable to perform our sharding algorithm as-is.

Before our sharding algorithm was implemented, we had a simple algorithm for obtaining data from MongoDB seen in Figure 5.

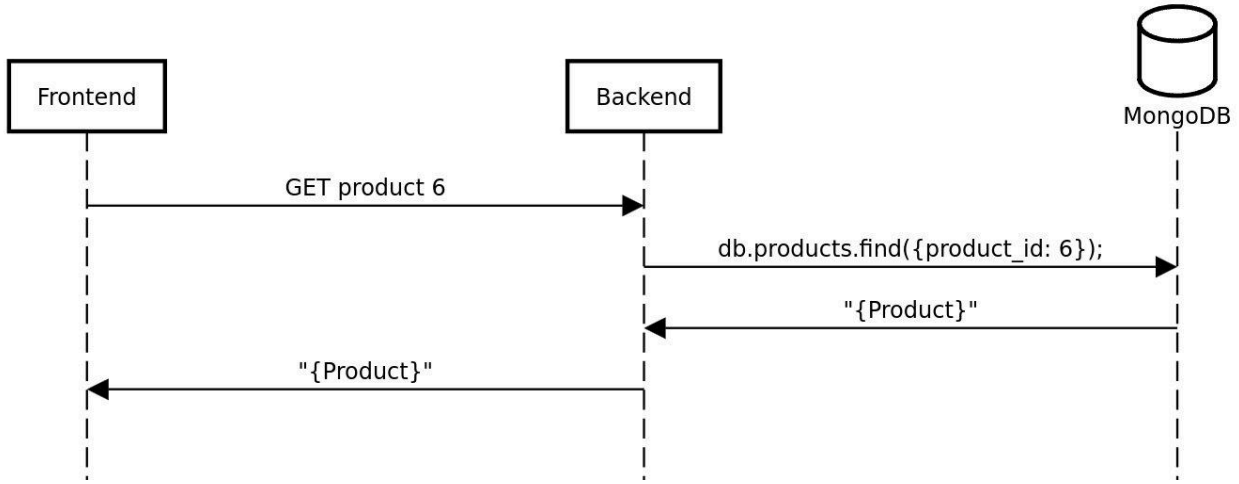

Figure 5: Old strategy for retrieving product data.

After the sharding algorithm, our strategy looked like Figure 6. We first must obtain the shard address from the Redis table, before being able to retrieve the data from MongoDB. If a connection to that database does not exist, we must also make a connection before retrieving the data. If the connection does exist, we will simply use the cached connection in our backend.

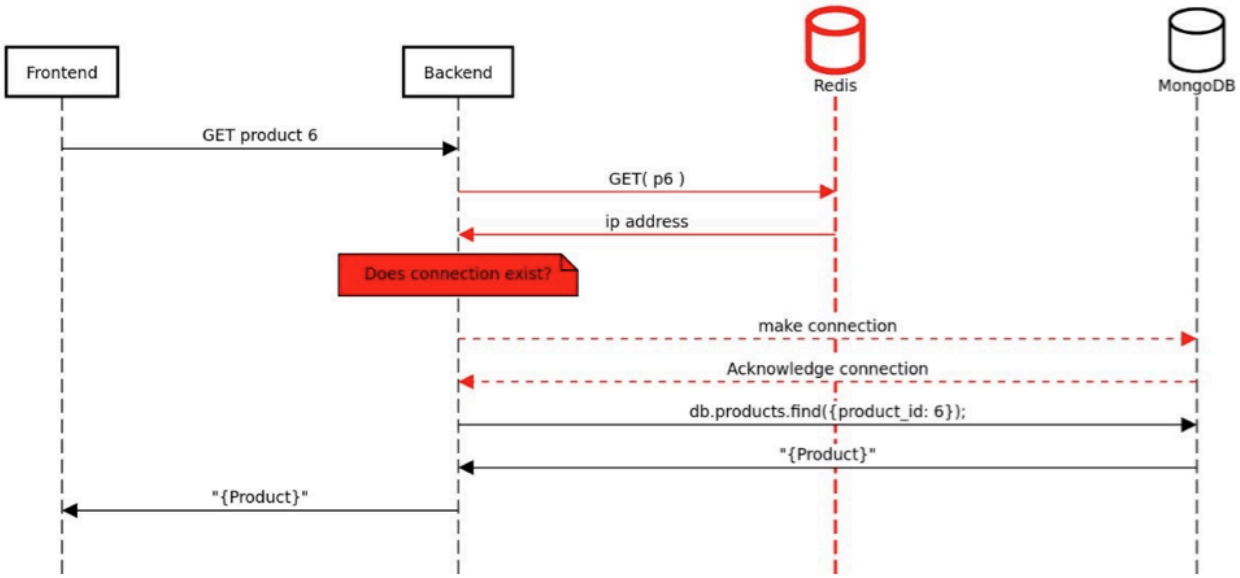

Figure 6: Sharding algorithm impact on GET product 6.

To mitigate the increased time it takes to retrieve a MongoDB entry, we also implemented a caching algorithm. Caching requests is a very common use case for Redis. With caching, we went with the following schema:

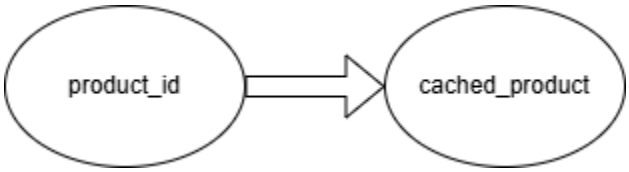

Figure 7: Redis caching schema

Our caching algorithm relies on two primary caching principles as listed in “Principles of Cache Design” (St. Michael, 2019). Temporal locality is the concept that a recently accessed piece of data is likely to be accessed again soon. We implement this by storing a product that is retrieved from MongoDB in the Redis cache for a set amount of time. The second principle is called spatial locality. This is the principle that nearby objects are also likely to be accessed soon. We implement this principle by adding the recommended items from Neo4j into the cache along

with the accessed product. While not based on a physical version of distance, the spatial locality in our caching algorithm refers to how close a user will be to clicking on a recommended product. This leads us to the final algorithm as shown in Figure 8.

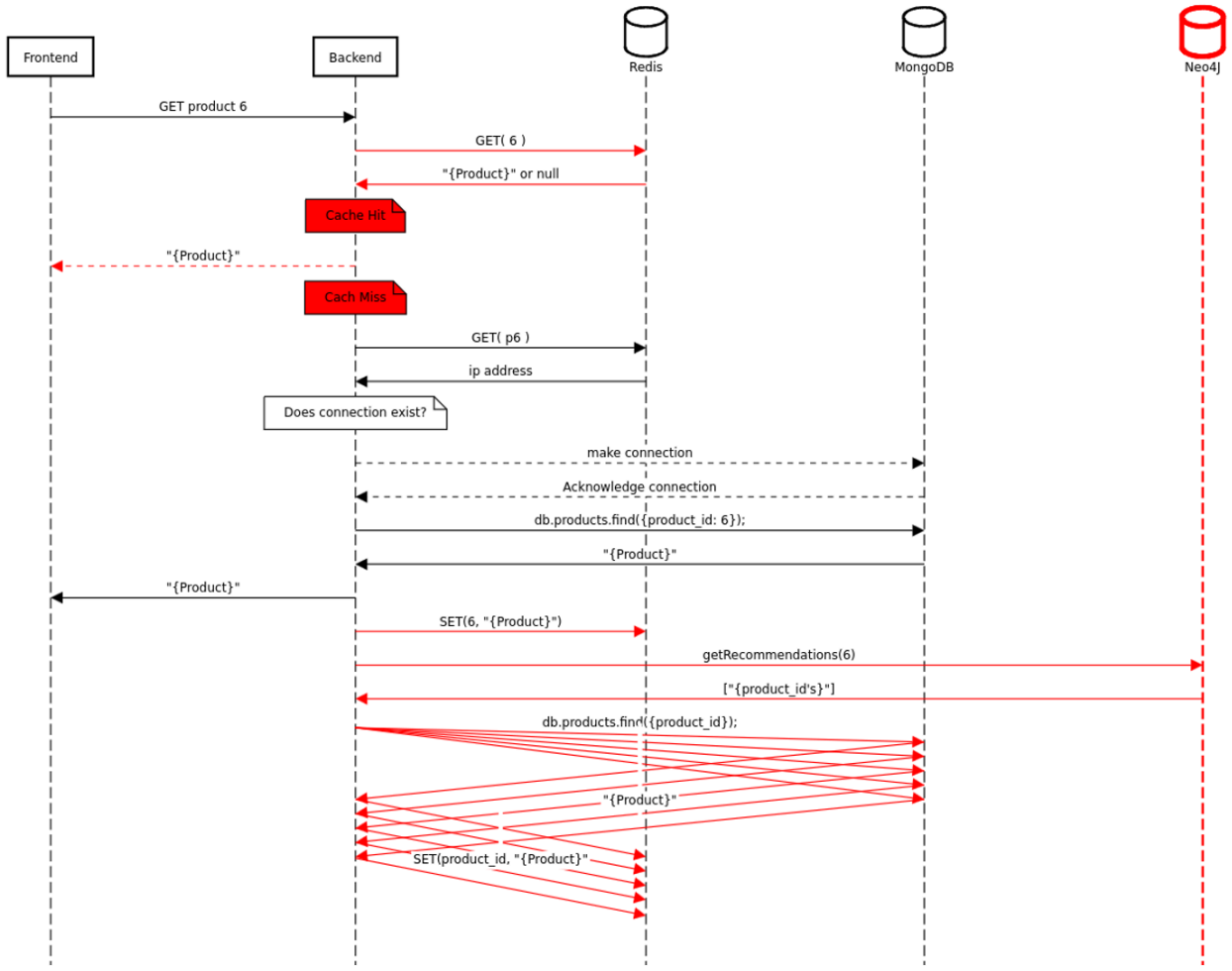

Figure 8: Example of sharding combined with caching for a GET request of product 6.

When a request is made to the backend for a product, we first check the cache. If the product id is stored in Redis, a cache hit, we return the cached product and finish. Otherwise, we perform our sharding algorithm, and afterwards, we put the product in the cache, along with up to five recommended products.

We decided on these particular systems for a variety of reasons. First of all, we have prior experience with both PostgreSQL and MongoDB. This made them more appealing than the other database systems, and they also offer both scalability and flexibility. The next component of the current polyglot is Neo4j, and it provides insight to relationships within the data. Neo4j has the ability to scale and is well known for its ease of use, which is why we chose it over other graph databases. The final component of the system is Redis, and it gives us the speed required to cache recently viewed data and to effectively shard across several machines. Redis provides increased efficiency to our system, especially as the data grows larger.

By using PostgreSQL, MongoDB, Neo4j, and Redis together, our polyglot system handles a wide variety of data and access needs efficiently. This approach allows us to pick the best tool for each job, ensuring we meet all the varied requirements of our application. With this setup, we achieve a well-rounded and high-performance data management solution that's tailored to the needs of our project.

The database systems are not the only technologies that needed to be considered for this project. Another important consideration was the language that was used, which was TypeScript. This language is compatible with the aforementioned databases, supports static typing, and offers advanced language features (interfaces, enums, generics, etc). For these reasons, Typescript is well suited for this polyglot system. On the backend, we used Node.js as the environment to run Typescript and Express.js as the framework for the API. The frontend used React and the Next.js framework to quickly create a web application, using the various API requests. Docker was another useful technology utilized, giving us the ability to package our libraries, tools, and code into containers that are maintained in isolated environments. By doing this, we were able to access the databases on different operating systems without error. We also used GitHub

throughout this project, giving us the ability to store and collaborate on code and other files. Each of these technologies provided essential structure to the project.

In the second semester, we incorporated the Jest unit testing framework. We decided to use Jest because it integrates well with both Javascript and React. Additionally, it offers minimal setup and configuration, provides built-in “mocking” to easily isolate specific functionality, and has snapshot testing. Due to time constraints, we only began testing the backend. With the tests we were able to finish, our coverage of the backend was approximately 40%. The unit tests revealed several problems that we were able to quickly fix. The first problem was when a user was created with a name larger than fifty characters the backend would crash. Our solution to this was putting a limit on the number of characters that we permit for this field. This problem persisted across several other input fields, which we fixed in the same way. This was the main problem we discovered with the limited unit testing we were able to perform, if we had more time we may have uncovered more problems.

Accomplishments

The following are the MoSCoW analyses for each of the semesters in our project. The boxes in gray indicate a feature we did not get to in that semester. Below is the MoSCoW for the fall semester.

MoSCoW (Fall 2024)			
Must	**Should**	**Could**	**Won’t**

At least 3 unique databases storing data for one application.	At least 4 unique databases storing data for one application.	A frontend to be able to have a way to interact with the data meaningfully.	A cloud database.
A backend which allows all databases to communicate.	Several datasets which can be used meaningfully with the Polyglot database.	A secure API (preventing injections).	Both a mobile app and a web app.
One dataset or use case to store and query in the databases.	An API for a frontend to interact with.	A full stack mobile app and web app.	A beautiful frontend.
A container for deploying the application across multiple devices (Docker).			

(Grey boxes indicate incomplete features.)

The main goals we established at the beginning of our project we were able to complete, along with several not as essential goals. Our polyglot service after the first semester consists of three databases, a backend to allow the communication of the databases, a use case, a Docker

container to deploy the application, API requests, and a semi-functional frontend. The backend is robust and able to handle the various needs required by the project. The main limitations of the project are the frontend, which has limited functionality.

A lot of time was required to become familiar with Promises and asynchronous execution in the Spring semester. When working with databases, performing a query takes an indefinite amount of time and during this period we do not want the program to be idle. Promises are used to let the program know that the information is guaranteed to be supplied eventually, allowing the program to move forward and perform other tasks until the query returns the information. As Promises are pipelined into other Promises, like when queries use results from other queries, this becomes increasingly more difficult to keep track of. To make this worse, the error messages that were displayed by problems caused by Promises were often vague and unhelpful. This forced us to go through line by line and carefully look at which Promises could have been causing problems. On the other hand, creating the connections to the databases was straightforward. Most of the time it was a simple function call to create a database client in the program. Once created, the client could easily perform queries on the specific databases with ease in our backend.

Next we have our MoSCoW analysis for the spring semester.

MoSCoW (Spring 2025)			
Must	**Should**	**Could**	**Won't**

A program to shard/distribute databases to more than one computer.	A frontend that demonstrates the application of each database.	Several datasets which can be used meaningfully with the Polyglot database.	A cloud database.
Implement Redis for sharding data.	Be able to shard with an arbitrary number of machines.		Both a mobile app and a web app.
A frontend to be able to have a way to interact with the data meaningfully.	A beautiful frontend.		
A container for deploying the application across multiple devices (Docker).	Create a test suite with at least 90% coverage of the backend.		
A secure API (preventing SQL injections).			

(Grey boxes indicate incomplete features.)

As we can see from the second semester MoSCoW analysis, we have met most of our goals for the project. However, there is still more work to do on this project which is discussed in the Future Work section. Our MoSCoW was chosen for items that we believe we could have accomplished with our time working on the project at WCU.

The items we did not get to, such as “Build a mobile app” was not in scope for this project and likely would not be continued in the future. However, “Several datasets which can be used meaningfully with the Polyglot database” is something that we could have chosen to do, but was not our main interest. In the end, we decided not to complete this one because our system was domain specific, and building a general system that would not allow us to account for tradeoffs in our data architecture. Just as Twitter optimizes for heavy writes when dealing with tweets, we would not have the necessary domain information available to optimize our program based on its use case.

Future Work

While this is the second semester of the project, and we have reached a point in which we are satisfied, there are still many avenues to explore in regards to the project. With regards to sharding, we would like to implement sharding on Postgres and Neo4j, and thus have a fully sharded application. Another feature to implement with sharding would be replicas, which is duplicating data in shards to have fault tolerance in the case of a system outage. This would ensure data is never lost and the app would continue to run during a failure of a database. Finally with shards, we could implement an automatic sharding algorithm, which is common in extremely large production systems. This would automatically spin up shards as needed, and then remove them when they are no longer needed. This could optimize space and RAM

efficiently so that no resource is wasted while still having the potential to handle billions of requests at a time.

We could also perform benchmarks on our polyglot solution. We could compare it to a single database solution to find the critical point where it becomes more beneficial to have a polyglot database or share a database over using a single database. It would also test the capabilities of our system. Another possibility is testing how many requests it can handle at a time, and truly define where its strengths and weaknesses lie.

Conclusion

In our first semester of our year long project, we were able to create a solid beginning to a polyglot distributed database service. We decided the most logical use case for this system would be to model an e-commerce application. To accomplish this, we have incorporated three databases so far, PostgreSQL, MongoDB, and Neo4J. These databases hold varying information about users, products, and transactions. Each of these databases have their own specific strengths which we are leveraging to create the most efficient product. Databases are difficult to understand without data present, so we are generating mock data. Also, we began working on a frontend that interacts with the polyglot service and allows individuals to experience the application in their web browser.

In the second semester of this project, we combined the final database, Redis, into our e-commerce app and focused on implementing sharding and caching. Continuing with the theme of an e-commerce app, we decided to shard our data to get our data physically closer to the end user of our product. This involved testing many different options, and finally settling on rolling out our own implementation using Redis. We also implemented caching as a way to optimize

query time for users interacting with our service. At the same time, we made significant improvements to our frontend by making it more visually pleasing and adding more functionality to view and understand the different schemas and data layouts behind our databases. Lastly, we began testing the code to ensure correctness of implementation and to eliminate bugs.

Works Cited

[1] Kraska, T., & Trushkowsky, B. (2013). *The New Database Architectures*. IEEE Xplore. https://ieeexplore.ieee.org/document/6509882

[2] MongoDB, "Database Sharding: Concepts & Examples," *MongoDB*. https://www.mongodb.com/resources/products/capabilities/database-sharding-explained

[3] Perkins, L., Redmond, E., & Wilson, J. R. (2018). *Seven Databases in Seven Weeks, Second Edition: A Guide to Modern Databases and the NoSQL Movement*. The Pragmatic Bookshelf.

[4] St. Michael, S. (2019, July 11). *Principles of cache design*. All About Circuits. https://www.allaboutcircuits.com/technical-articles/principles-of-cache-design/

[5] Wade, B., & Chamberlin, D. (2012). *IBM relational database systems: The Early Years*. IEEE Xplore. https://ieeexplore.ieee.org/document/6297962