Investigation of System Architecture, Databases, and Caching systems For backend web development

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Bachelor of Arts in the
MCS at The College of Wooster

by
Jungho Park

The College of Wooster
2026

Advised by:

Drew Guarnera (Computer Science)

THE COLLEGE OF
WOOSTER

Abstract

Include a short summary of your thesis, including any pertinent results. This section is *not* optional, and the reader should be able to learn the meat of your thesis by reading this (short) section.

This work is dedicated to the future generations of Wooster students.

Acknowledgments

I would like to acknowledge Prof. Lowell Boone in the Physics Department for his suggestions and code.

Vita

Publications

Fields of Study Major field: Major

Minor field: Minor

Specialization: Area of IS research

Contents

List of Figures

Figure Page

List of Tables

Table Page

List of Listings

Preface

The purpose of this document is to provide you with a template for typesetting your IS using LaTeX. LaTeX is very similar to HTML in the sense that it is a markup language. What does this mean? Well, basically it means you need only enter the commands for structuring your IS, i.e., identify chapters, sections, subsections, equations, quotes, etc. You do not need to worry about any of the formatting. The `woosterthesis` class takes care of all the formatting.

Here is how I plan on introducing you to LaTeX. The Introduction gives some reasons for why one might find LaTeX superior to MS Word™. Chapter 5 will demonstrate how one starts typesetting a document and works with text in LaTeX. Chapter **??** discusses the creation of tables and how one puts figures into a thesis. Chapter **??** talks about creating a bibliography/references section and an index. There are three Appendices which discuss typesetting mathematics and computer program code. The Afterword will discuss some of the particulars of how a LaTeX document gets processed and what packages the `woosterthesis` class uses and are assumed to be available on your system.

Hopefully, this document will be enough to get you started. If you have questions, please refer to **mgbcr04**, **kd03**, **ophs03**, **feu02**, **fly03**, or **gra96**.

CHAPTER 1

Introduction

So why would you want to use LaTeX instead of Microsoft Word™? I can think of several reasons. The main one for this author is that LaTeX takes care of all the numbering automatically. This means that if you decide to rearrange material in your IS, you do not have to worry about renumbering or references. This makes it very easy to play around with the structure of your thesis. The second reason is that it is ultimately faster than Word™. How? Well, after a week or so of using LaTeX , you will begin to remember the commands that you use frequently and won't have to use the LaTeX pallet in TeXShop or TeXworks. So, you can just type everything including the mathematics, where with Word™ you would have to use the Equation Editor.

I have also tried to make things more efficient by organizing the example folder as follows. There is a `main.tex` file which is what you will enter all the information about your IS into and is the document you will typeset. `main.tex` also has explanations about other files that you might need to edit. In addition, there are folders for chapters, appendices, styles, and figures. This structure is there to try and reduce file clutter and to help you stay organized. There should also be a .bib file which you can use as a model for your own .bib file. The .bib file has your bibliographic information.

LaTeX is easy to learn. For an average IS, the author will only need to learn a handful of commands. For this small bit of effort, you get a tremendous amount of flexibility and a very beautiful document. The following chapters will introduce some of the common things a student might need to do in a thesis.

1.1 What is in `main.tex`?

Before we move on let's talk a little bit about what is at the beginning of `main.tex`. The file starts with `\documentclass{woosterthes` which must be at the beginning of every IS. In the brackets are options for the woosterthesis class. The options are the same as for the `book` class with some additional options `abstractonly`, `acs`, `alltt`, `apa`, `blacklinks`, `chicago`, `citeorder`, `code`, `colophon`, `dropcaps`, `euler`, `foreignlanguage`, `guass`, `index`, `kaukecopyright`, `maple`, `minimalstyle`, `mla`, `nostyle`, `palatino`, `picins`, `scottie`, `singlespace`, `tikz`, `verbatim`, `wblack`, and

`woostercopyright`. If no options are specified then the class default options `letterpaper`, `12pt`, `oneside` , `onecolumn` , `final`, and `openany` are used.

The `abstractonly` option will allow you to print just the Abstract. The `acs` option implements the American Chemical Society citation and reference style. The `alltt` option loads the `alltt` package for using typewriter type in various ways and the `apa` option implements the APA citation and reference style. The `blacklinks` option will make the hyperlinks in the PDF version of the thesis black and suitable for printing; normally the links are colored to provide visual clues to the reader. The `chicago` option implements Chicago style citation and references. The `citeorder` option orders the references according to the citation order of the IS. The `code` option will use `listings` style to format program code examples. The `colophon` option will include a colophon which is a section that describes the fonts and other settings used to produce the manuscript. `dropcaps` loads the `lettrine` package for doing dropped capitals and the `euler` and `guass` options load the `woofncychap` package with the named option which will change the look of chapter headings. The `foreignlanguage` option will load the `csquotes` package and either the `polyglossia` or `babel` package depending on if XeTeX is being used to allow the input and formatting of sections of text in a foreign language. The `index` option will allow the `makeidx` package to be loaded so that if you have index entries they will be added to an index (this reqires additional steps). The `kaukecopyright` option will put the Kauke Hall symbol with the pre 2021 wordmark on the copyright page. The `maple` option will load the Maple package for including Maple code. The `minimalstyle` option will use custom styling for the Table of Contents and List of Figures, Tables, and Listings and otherwise revert to the default `book` class styles. The `mla` option implements the MLA citation and reference style. The `nostyle` option will remove all custom styling and revert to the default `book` class styles. The `palatino` option will use the `pxfonts` package which uses the Palatino fonts. The `picins` option will use the `wrapfig` package to allow text to wrap around images and `verbatim` allows one to set verbatim what is entered. The `tikz` option loads the `TikZ` package enabling users to draw figures in their LaTeX document. The `singlespace` option will use the `setspace` package to typeset the document in singlespace. The `wblack` option includes an opaque Wooster ”W” (as of 8/2021) in the background of the title page instead of the default opaque Kauke Hall image, the `scottie` option includes an opaque grayscale image of the Scottie mascot in the background of the title page instead of the default opaque Kauke Hall image, and the `woostercopyright` option includes a copyright notice with the new (as of 8/2021) Wooster wordmark (see Appendix **??** for images of what these options produce). Adding or deleting options from the comma separated list will change the appearance of the document and some options should only be used after consulting your advisor. Now let’s move on to some other things that you’ll need to deal with: text, figures, pictures, and tables.

CHAPTER 2

Theoretical Foundations and Literature Review

2.1 Backend Software Design and Architecture Principles

There are multiple architectural styles that could be applied when designing the backend system of a web application. The two main categories are monolithic and distributed. Subcategories such as layered services, microservices, event-driven services, are all under these two architecture styles. Each of these design have the pros and cons.

2.1.1 Monolithic

A monolithic application is where all the logical components of the application is deployed as one unit. This also means that the application will be ran in one process. For example, if there is a stand-alone Python Flask application deployed to a server, this is monolithic applications. The general pros and cons of this architecture style are the following:

Pros: - Simplicity: Typically, monolithic applications have a single codebase, which makes them easier to develop and understand - Cost: Monolithis are cheaper to build and operate because they tend to be simpler and require less infrastructure. - Feasibility: Monoliths are simple and relatively cheap, freeing developers to experiment and deliver systems faster. - Reliability: Monoliths makes few or no network calls, which usually means more reliable application. - Debuggability: If a bug is spotted or get an error stack trace, debugging is easy, since all the code is in one place. Cons: - Scalability: - Evolvability: Making changes to monolithic applications become harder as it grows. Since the whole application is one codebase, it is not possible to adapt different technology stacks to different domains if needed. - Reliability: Since monolithic applications are deployed as a single unit, any bug that degrades the service will affect the whole application. - Deployability: Implementing any change will require redeploy8ing the whole application, which could introduce a lot of risk.

2.1.1.1 Layered Architecture

Layered architecture is a sub category of the Monolithic architecture. Applications that are designed with this architecture has three layered parts: the presentation layer, the workflow layer, the persistence layer, and the database layer. Each layer has it's own tasks and separated within one application. Presentation Layer: The presentation layer is where the UI is displayed and where the users interact with the system. All components that are related to the UI will be included in this layer. Workflow Layer: The workflow layer consists all code that are related to logic such as business logic, workflows, and validations. This is where most of the application's code is contained. Persistence Layer: The persistence layer encapsulates the behavior that is needed to make objects persistent, such as mapping the architecture to code-base hierarchies into set-based relational databases. Database Layer: The database layer is optional. This layer includes the database, or some kind of way to persist information.

Each of the layer may contain multiple problem domains. For example, for a restaurant, there could be domains like place order, deliver order, manage recipes, or mange inventory. Different problem domains could exist together in each layer of the application.

An implication of the layered monolithic architecture is the MVC design pattern. In MVC, the model represents business logic and entities in the application; the view represents the user interface; and the controller handles the workflow, stitching model elements together to provide the application's functionality, as shown here: (head first software architecture chapter 6)

General Advantages of Layered Architecture: - Feasability (Simplicity) - Technical Partitioning - Data-intensive - Performance (if well designed) - Quick to build - General Disadvantages of Layered Architecture - Deployability - Complexity - Scalability: Non flexibility in change in problem dodmain - Elasticity - Testability Some applications using the layered architecture may have different structure from the original. Some layers might be separated, such as the presentation layer being separated from the other layers. Each structure will have advantages and disadvantages within the layered architecture.

2.1.1.2 Modular Monolith

Modular monolith is when an application is divided into different modules within the monolithic system. A module is one domain of the application, and each domain will have its independent code base. For instance, an application for a restaurant might have different domains such as orders, deliveries, or recipes. In this case, there could be separate databases for each modules, or multiple schemas within a single database. It is important to keep in mind that only the code base is independent. Pros: - Domain partitioning: each domain will be implemented in a different module, which allows to build teams that specializes in one or more of these domains. - Performance: Since there are no network calls within the application like other monolithic applications, performance is good. - Maintainability: Each module

are separate from each other. - Testability: Cons: - Hard to reuse: Since each module have its own code, it is hard to reuse the logic and utilities across the modules - Single set of architecture: Since all the code is within one application, logics and databases could become complicated, making it hard to scale.

2.1.1.3 Microkernal Architecture

The microkernel architecture is consisted with a core of the application, and plugins. The core of the application is where the main application is being run, and the plugins are connected to the core application. It is used when a lot of customization is needed for the software

2.1.2 Distributed Systems

Distributed architecture is when the logical componnents of the application are split up into multiple units. These units each run in their individual process and communicate with each other over the network. This architecture style encourage loose coupling of each services. The general pros and cons of this architecture style are the following:

Pros: - Scalability: Distributed architectures deploy different logical components separately from one another, so it is easy to add new services. - Modularity: Distributed architecture encourage a high degree of modularity because their logical components must be loosely coupled. - Testablility: Each deployment only serves a select group of logical components. This makes testing a log easier-even as the application grows. - Deployability: Distributed architectures encourage lots of small units. They evolved after modern engineering principles like continuous integration, continuous deployments, and automated testing became the norm. - Fault Tolerance: Even if one piece of the system fails, the rest of the system can continue functioning. Cons: - Performance: Distributed architectures involve lots of small services that communicate with each other over the network. This can affect performance, but there are ways to improve this. - Cost: More servers are needed to deploy multiple units. These services will need to talk to each other, which entails setting up and maintaining network infrastructure. - Simplicity: Distributed systems are complicated to understand from how they work to debugging errors. - Debuggability: Errors could happen in any unit involved in servicing a request. Since logical components are deployed in separate units, tracing errors could become complicated.

There are both pros and cons for monolithic and distributed systems and choosing the appropriate architecture for each situation is crucial. After deciding which of the two main architecture will be used, the subcategories shows the more detailed ways of designing and structuring each units in the application(s).

2.1.2.1 Microservices Architecture

A microservice is a service that is separately deployed unit of software that performs some business or infrastructure process.(chapter 10). A microservices architecture is part of the layered system, where microservices communicate

which each other to make an application. Since the system is divided into multiple parts, it is essential how to divide the application, such as deciding how small or how big each microservices could get into. Some of the factors that is considered to make microservices smaller are the following: cohesiveness, fault tolerance, access control, code volatility, and scalability. If a part of the software has lack of cohesiveness and loosely coupled, it is a good idea to separate it into smaller microservices. This will allow higher scalability. If a certain part of the application produces fatal errors, having those part in a sparate microservice will decrease the probability to shut down the whole system. For security and authentication, it is important to have these access controlabilities into a single microservice, so it does not get too complicated when managing the information. Finally, if one part of the microservice change, or scale faster than the others, it is good to consider to have a separate microservice for that application, since testing the entire microservice would be much more challenging compared to a small portion of the microservice. On the other hand, there are times when it is encouraged to make the microservices bigger: Database transactions, data dependencies, and workflow. It is not possible to perform a single database commit or rollback when a request involves multiple microservices. For data consistency and integrity, it is important to combine functionalities that require these kind of behavior into a single microservice. If a part of a microservice has highly coupled data, such as when a database table refers to the key of another database table, it is better to keep these functionalities as a single microservice, to keep the data integrity of the database. If a single request requires separate microservices to communicate with each other, this request is coupled. If too much coupling is occurred between microsesrvices, there are many negative effects. For example, performance is affected by network, security, and data latency. Scalability is affected because each microservice in the call chain must scale as the other microservices scale (something that is hard to coordinate). Fault tolerance is affected because if one of the microservices in the chain becomes unresponsive or unavailable, the request cannot be processed. It is good practice to consider the workflow and decide whether to keep the microservice big.

Balance

Sharing functionalities

There are many times when the same code has to be used in multiple microservices. There are mainly two ways for sharing code when building a microservice architecture. Creating a shared service, or a shared library. A shared service is a separate microservice that contains a shared functionality that other microservices can call remotely. The advantages of using this is that eventhough a code is changed in the shared microservices, code in other microservices are not required to change. Also, this shared service could be written in any language, which is useful when microservices are implemented in multiple languages. A disadvantage of using a shared service is coupling that happens between the microservices and the shared service. This leads to risks when changing a shared service since it can affect the other microservices that call it. Furthermore, when the shared service is down for some reason, the microservices that require the shared service would not function. Another disadvantage is network latency, …. A shared library is a more com-

mon way for code reuse. A library will be built including all the code that are reused in different microsesrvices, and once they microservices are deployed, each microservice will have all the shared functionality available. The biggest advantage of using a shared library is that network performance and scalability is better than shared service, since it is not remote, but code is included in the compile time of the microservice. However, multiple shared libraries will be needed if microservices are written in different programming languages. Also, managing dependencies between microservices and shared libraries could be a challenge if there are multiple microservices using the shared code.

Workflow

Microservices communicate with each other to make the whole application. It is important to know how these are connected, and this is where workflow comes in. The term workflow means when two or more microservices are called for a single request. There are two ways to handle workflows: centralized workflow management, or decentralized workflow management.

Centralized workflow management:Workflow

This workflow management style is where there is a microservice that coordinates all the microservices that are needed to handle a certain single request. This microservice will be responsible for calling all related microservices, knowing the current state of the workflow and what happens next, summarize all data from each microservice, and handling errors. The advantage of this workflow management style is that the order of microservices for each requests are clear. The central microservice always has the exact routes the request hast to take, which allows to track the status where each requests stopped and where to restart. This allows to handle errors efficiently. Also, it is easy to change a workflow since this all changes in one central microservice The disadvantage is that tight coupling between the central microservice and the other microservices. This can lead to lower scalability, since changing a microservice will affect the central microservice. Moreover, performance might get delayed since the central microservice is indeed another microservice that requires remote calls and it saves the workflow state data in a database which slows down the performance.

Decentralized workflow management

The decentralized workflow does not have a central microservice, but rather all microservices redirect to another microservice by the given request until the request is complete. It is important to not use one of the microservices as a central microservice. The advantage of using this workflow pattern is that it is loosely coupled compared to the central workflow management, which has better scalability. Also, since the microservices does not have to connect back to the central microservice, it has better responsiveness, meaning less delay and better performance. On the other hand, this pattern lacks in error handling, since each microservice is responsible for managing the error workflow, which could lead to too much communication between services. Also, because there is not a central microservice that has the order of microservices that needs to be called, it is hard to tell which state the response is at, which decreases the ability to recover when the request is delayed and needs to be restarted again.

Pros and Cons of microservice architecture. There are moments where the microservice architecture are needed when choosing the software design. There are multiple advantages when choosing this structure. Loose coupling is a huge advantage. Since each microservices are single-purpose and they are deployed separately, and easy to maintain. This means that it is easy to change a particular function that needs modification. Also, it is easy to test the application, as the scope is much smaller compared to the monolithic architecture. Fault tolerance is another advantage that comes from loose coupling. Even if a particular microservice fails, it does not break the whole system. Furthermore, microservice architecture is easy to scale and evolve since we just have to add or modify a microservice related to the area.

However, there are also disadvantages of using the microservice architecture. This architecture is complex. It requires multiple decisions(the aspects discussed above: workflow, shared code etc..) depending on the situation. Since it is complex, as microservices communicate with each other, the performance also decreases. The request might have to wait for the network or undergo additional security checks and make extra database calls. Lastly. deploying all of these microservices will increase the cost of the entire application.

2.1.2.2 Event-Driven Architecture(EDA)

What is an Event An event in computer science is a way for a service to let the rest of the system know that something important has just happened. In EDA, events are the means of passing information to other services (head first chapter11). The event contains data and it is broadcasted to services using topics that are connected to the sender of the event. However, events are asynchronous, which means that although the services are connected, the sender of the event does not wait on the response of the receiving service. This is the key difference between a event and message. Messages includes a command or some kind of a request for the receiving service, and the service sending out the messages require a response, making it synchronous. Messages are also sent to only a single service using queues.

Asynchronous vs Synchronous Asynchronous communications do not need to wait for the response, even though the receiving services are available or not. These kinds of communications are also called Fire and Forget. On the other hand, synchronous communication needs to stop and wait until the response, which means that the service in response must be available. EDA relies on the asynchronous communication when sending and receiving events.

Advantages, Disadvantages of Asynchronous Communications Asynchronous communications has advantages in responsiveness compared to synchronous communications. Since responses by the receiving services are unnecessary, it takes less time to complete a request. However, this is also a crucial disadvantage. Since we do not know if the receiving services have successfully completed the request, it is prone to error handling. Here is a diagram.

Advantages of EDA Event Driven Architecture is highly decoupled, which makes all services independent and easy to maintain. Furthermore, the asynchronous communication increases the performance of the application. Since

EDAs are highly decoupled with this type of communication, it is easy to scale and evolve. Disadvantages of EDA Similar to microservices, EDAs are complex. Deciding which database topology for the architecture, asynchronous communications and parallel event processing makes adds complexity to the application. Also, asynchronous communication makes it more difficult to test out the program. Since the request is proceeded without any response or synchronous calls, the context of the test is vague. If the application needs multiple synchronous calls, EDA is not the right choice for the product.

2.2 Databases

2.2.1 Types of Databases

There are mainly two types of database that are used in software engineering: Relational database and non-relational databases. These two databases defer by how they store and data, which influences different aspects if databases including the structure, data integrity mechanism, performance and more.

2.2.1.1 Relational Databases

Relational Databases store data in tables by columns and rows, where each column represents a specific data attribute, and each row represents an instance of that data(aws document). Each table must have a primary key, which is an identifier column that identifies the table uniquely. The primary keys are used to establish relationships between tables, by using the related rows between tables as the foreign key in another table. Once two tables are connected, it is now possible to get data from both tables in a single SQL query.

Advantages and Disadvantages: Relational databases follow a strict structure, which allows users to process complex queries on structured data while maintaining data integrity and consistency. The strict structure also follows the ACID (atomic, consistency, isolation, and durability) properties for enhanced data integrity. However, because of the rigid structure, it is hard to scale compared to nonrelational database.

2.2.1.2 Non-Relational Databases

Nonrelational databases means that there isn't a schema to manage and store data. This means that data does not require constraints that the relational database required, such as fixed schema, primary key constraint, or not null constraints. This allows more flexibility in the structure and size of the database, or anything that may change in the future. There are different types of non relational databases: Key-Value databases, Document database, and Graph database. Key-Value database store data as a collection of key-value pair, where the key is served as the unique identifier. Both the key and values could be anything as objects or complex compound objects. Document databases are used to store data

as JSON objects. Since it is readable by both human and the machine, it has the ease of development. Lastly, there are graph databases, where they are used when a graph-like relationships are needed. Unlike the relational databases, which store data in rigid schema, graph databases store data as a network of entities and relationships, providing more flexibility to anything that is prone to change.

Advantages and disadvantages: Nonrelational databases have a less rigid structure allowing more flexibility. This is useful when the data changes requirements often. For example, when a specific table needs to changes in columns, this would be hard to preform because other tables might be associated with the specific column. However, nonrelational databases are not constraint to fixed schema, which allows easy changes on specific columns or unique identifier. Performance is another strength of nonrelational database. The performance of these databases depend on outer factor such as network latency, hardware cluster size, which is different from relational databases which depends internal factors such as the structure of the schema. However, because of the flexibility in the structure, data integrity is not always maintained. Consistency is an issue since the state of the database changes over time as structures might not be consistent.

2.2.2 Normalization

Compared to a nonrelational database, relational databases benefit from integrity, with the process of normalization. The definition of normalization is a way of organizing data in a database. During this process, redundant data will be reduced to improve data quality and optimize database performance. There are types of normalization such as 1NF and 2NF, and as the normal form gets higher, it is a better design of the relation. However, most databases tend to be needed until the third normal form, which avoids most of the problems common to bad relational designs.

The Second Normal Form(2NF) The two criterias that 1NF meet are the following:

1. The data are stored in a two-dimensional table.
2. There are no repeating groups. (rddI chapt 7)

Here the repeating groups mean that if an attribute that has more than one value in each row of a table. (relational database design and implementation 4th) For instance, if there is a column that requires more than one value, such as items (instead of item), this will be a repeating group. (show table) Having three or more dimensions or having repeated groups for a single table results in more complexity and difficulties when querying the database. This is why we need the 1NF normalization.The 1NF is the most simple normalization that could be done in a relational database and it is pretty clear. However, the first normalization form is not enough from benefiting from using relational databases.

Problems with 1NF

Data could become redundant even though repeated rows are deleted. For example, If there is a table called students with columns(id, name, birthday, course id, course name), every time the same student adds a class, their name will be

repeated in each record. Furthermore, there are anomalies in update, insertion, and deletion. Making an update to one of the records (student name) would require to update other records that are associated. Missing out any of the records with the student name will result in data inconsistency. Inserting a new entity could be difficult unless there are related data. For instance, if a new course is to be added to the table, this would be unavailable until there is a student taking the course since the primary key will be the student's id, and there could not be a row without a primary key. Deletion could also be a problem when the row deleted contained the last data for a certain column. For example, if one student is taking a class called CS200, and that record is deleted, the table would not have the information that the class CS200 exists. To solve these problems, we must use higher levels of the normalization form.

The Second Normal Form(2NF)

The two criterias that 2NF meet are the following:

1. The relation is in first normal form
2. All nonkey attributes are functionally dependent on the entire primary key.

Functional dependency is the key term in the 2NF. A functional dependency is a one-way relationship between two attributes, such that at any given time, for each unique value of attribute A, only one value of attribute B is associated with it throughout the relation (RDDI chapt7). In other words, this means that all other columns except the columns of the primary key or keys, are dependent on the primary key or the candidate key.

By using the functional dependencies, we could create the second normal form relations. After analyzing the functional dependencies, primary keys would be decided. It is common to decide attributes that have dependencies for the primary key of the table, and the all the other attributes will be the non-key attributes. For example, going back to the student table example, Student name, and birthday would be dependent on the student id, so the student id will be the primary key. Courses the student takes does not depend on the student id, but rather on the course id. We could create another table using the course id as the primary key, and the course name as the functional dependencies. However, the courses the student takes will be dependent on the courses, so there will be a foreign key to construct that relationship. So far, we can identify the relationship of the tables as the following:

So far, we can identify the relationship of the tables as the following:

- **Student**(*student_id* (PK), *course_id* (FK), *name*, *birthday*, *room_num*)
- **Courses**(*course_id* (PK), *name*)

We are assuming that each student can only take one course.

The relationship can be represented as:

$$\text{Student(course\_id)} \rightarrow \text{Courses(course\_id)}$$

By doing this, some of the problems from the first normal form are solved. The functional dependency solves the insertion, deletion, and update anomalies. We can now insert a new course into the course table without needing to insert a student at the same time, delete a student's course without losing the record of the course itself, and update course information in one place instead of multiple rows, which preserves data integrity. This will provide more stable and consistent database design compared to 1NF.

Problems with 2NF

Although some of the problems with 1NF were solved, there are still anomalies to be resolved. Insertion, deletion, and update anomalies still exist. For example, let us assume we have the following dependencies:

- **Student**(*student_id* (PK), *course_id* (FK), *name*, *birthday*, *room_num*)

$$\text{Student}(\text{course\_id}) \rightarrow \text{Course}(\text{course\_id}, \text{name}, \text{room\_num}(FK))$$

- **Course**(*course_id* (PK), *name*, *room_num* (FK))

$$\text{Course}(\text{room\_num}) \rightarrow \text{Room Number}(\text{room\_num}(PK), \text{name})$$

- **Room Number**(*room_num* (PK), *name*)

Even though the tables are in 2NF, insertion, deletion, and update anomalies can still occur due to these transitive dependencies.

The functional dependency Course -> Room Number introduces anomalies that still remain in 2NF. For example, we cannot record a student's enrollment in a course if the course's room number has not been decided, which creates an insertion anomaly. Also, if we delete the last student enrolled in a course, we also lose the record of the room number for that course, which is a deletion anomaly. Finally, if a course's room number changes, we must update it in every student's record for that course, creating an update anomaly. These problems occur because the room number depends on the course (a non-key attribute) rather than directly on the student ID, and they are resolved by moving the design into 3NF.

The Third Normalization Form(3NF) The two criterias that 3NF meet are the following:

1. The relationship is in second normal form.
2. There are no transitive dependencies.

Transitive dependencies Transitive dependencies exist when the following functional dependency pattern occurs:

$$A \rightarrow B \text{ and } B \rightarrow C, \text{ therefore } A \rightarrow C$$

(Chapter 7).

This is the same type of relationship where we had problems with 2NF. For example:

- **Student**(*student_id* (PK), *course_id* (FK), *name*, *birthday*, *room_num*)

$$\text{Student}(\text{course\_id}) \rightarrow \text{Course}(\text{course\_id}, \text{name}, \text{room\_num}(FK))$$

- **Course**(*course_id* (PK), *name*, *room_num* (FK))

$$\text{Course}(\text{room\_num}) \rightarrow \text{Room Number}(\text{room\_num}(PK), \text{name})$$

Going back to this example, we can see that a non-key attribute in the **Student** table (*room_num*) depends on another non-key attribute (*course_id*).

To remove transitive dependencies, we should break the relations into separate tables. In this case, we can:

- Remove the *room_num* column from the **Student** table to remove the relationship between non-key attributes.
- Alternatively, make the second determinant in a table a candidate key so that no non-key attribute depends on another non-key attribute within the same table.

Applying either method will resolve the anomalies we encountered in the second normalization form (2NF).

2.2.3 Bad Designs

There are multiple ways how relational databases could be badly designed. Bad designs will not benefit from data integrity, which is one of the advantages of relational databases.

There are mainly three problems related to bad design of relational databases:

1. Unnecessary duplicated data and data consistency
2. Data Insertion Problem
3. Data Deletion Problem

2.2.4 Querying and Performance

2.2.4.1 N+1 Query Problem

2.2.4.2 Performance Related Strategies

A. Indexing B. Sharding C. Partitioning

2.3 Caching

2.3.1 Client Side Caching

CDN HTTP Caching

2.3.2 Client Side Caching

Redis

CHAPTER 3

Implementation

3.1 General Design Choices Made

3.1.1 Backend Design Choices

3.1.2 Frontend Design Choices

CHAPTER 4

Results

CHAPTER 5

Challenges and Lessons Learned

5.1 What worked well

5.2 What did not work well