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)

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

List of Tables

List of Listings

CHAPTER 1

Introduction

My first web application was a Flask application with a simple backend, frontend, a database connected to it, and a few API calls. I thought that web development was simpler than it sounds. However, I changed my mind within a few months of learning the frontend and the backend side of web applications. As I worked on projects, new problems arose as I try to make the scale of the project larger. Also, depending on the project, I had to learn new strategies and technologies to achieve the goal I wanted. More things were there to learn, and I started realizing that several design choices could be made in every situation. Not being able to answer "why this specific choice?" for some of the choices I made realized that I do not have a full understanding of the decisions that could be made in web development, and led me to research this topic.

Covering all the topics in web development was the primary goal, but it could be a lot to process a deep understanding in one year of the Independent Study period. I narrowed down my subjects to target the topics I encountered the most and personal interests, with the help of the backend roadmap [1]. Software architecture choices were a personal choice of interest. Although most of my applications were monolithic due to a smaller scale, I wanted to dig deeper into the structures of monolithic applications and also distributed systems, so that I could apply the knowledge when I start to work with larger-scale applications.

The second topic is database systems. I have taken the database class during my sophomore year, but web development required much deeper knowledge. Types of databases to choose, ways to structure the database for performance and scalability were always a confusion for me. To fully understand the decisions I am making, I researched this topic to be able to make logical decisions in the future when creating applications.

The last topic is about caching systems. There were multiple times in my past projects where the application was slow due to database queries or functions to fetch data from external APIs. Caching is one of the solutions to solve this problem, and it will be discussed throughout this project. Caching data that calls database queries or API calls often makes the application significantly faster, and I want to investigate which strategies and types of caching systems are sufficient in every scenario.

A web application that uses these three technologies is the best way to show that I have a full understanding of the choices I have made. I developed a local cuisine recommender web application to show the knowledge and research done in these areas.

The motivation for creating the application arises from the discomfort of finding local cuisines when traveling to foreign countries. It is a good experience to try out the local cuisines of the destination area, especially for people who love food. However, searching directly on Google Maps does not accurately show the famous dishes in the specific region, but rather recommends restaurants. Going straight to the restaurants sometimes works, but sometimes it is helpful if there is a set of cuisines to try, already layout and choose the restaurant where the cuisines are available. This needs a two-step search: first on a search engine such as Google to find the famous food in the area, and second, searching the name of the dish on Google Maps. The purpose of the Local Cuisine application is to make the process of searching restaurants more efficient.

CHAPTER 2

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 their advantages and disadvantages.

2.1 Monolithic

A monolithic application is where all the logical components of the application is deployed as one unit, which also means that it will be run in one process. For example, if there is a stand-alone Python Flask application deployed to a server, this is monolithic application. This means that the UI components should also be included in the same unit. If there is a separate frontend server, this is not a monolithic application anymore. There are multiple reasons to build a project with the monolithic architecture.

One of the benefits of the monolithic architecture is its simplicity. Since a typical monolithic application have a single codebase, it is easier for the developers and collaborators to understand the overall structure. Even when a developer joins in the middle of the development process or reading an older code base, understanding program will be generally easier compared to distributed systems structure. Another advantage of using the monolithic structure is the cost. Since only one unit has to be deployed, which is simple and requires less infrastructure, it is cheaper to operate overall. Because of the cheap cost, developers are able to experiment and deliver systems faster, which is beneficial in a fast pace environment. Reliability is another aspect, where monolithic applications make few or no network calls, which usually means more reliable application. Lastly, it is easier to find bugs when spotted compared to distributed systems since all the code is in one place.

On the other hand, there are also disadvantages of using the monolithic architecture. As the monolithic application grows, the application becomes harder to scale. Since all the code is in one codebase, it is not possible to adapt different technology stacks to different domains if needed. This also connects to less reliability. A single bug or error occurred in the application could degrade the service and affect the whole application. Lastly, when deploying the application, implementing any change will require redeploying the whole application, which could introduce a lot of risk.

2.1.1 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 its own tasks and separated within one application.

- **Presentation Layer:** The presentation layer is where the user interface (UI) is displayed and where the users interact with the system. All components that are related to the UI are included in this layer.
- **Workflow Layer:** The workflow layer consists of all code 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 required to make objects persistent, such as mapping the architecture or code-base hierarchies into set-based relational databases.
- **Database Layer:** The database layer is optional. This layer includes the database or any mechanism used to persist information.

Each layer may contain multiple problem domains. Let's take an application for a restaurant as an example. There could be domains such as 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, which is an abbreviation for model view and context. 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.

The layered architecture has several advantages. One of them is the feasibility of the application. Since the structure is organized into distinct layers with specific responsibilities, it is clear for developers to understand the overall structure of the code. This separation also allows technical partitioning, which makes it much easier to manage and maintain big projects. The division of each section decreases the possibility of bugs, which could be a crucial disadvantage of monolithic applications. Furthermore, the layered architecture is quick to build particularly among the projects that have clear relationships among the components, as different development teams can work on different layers in parallel.

There are also disadvantages to consider. The layer could add complexity to the application compared to a simple monolithic application. However, this would still be less complicated compared to the complexity of the distributed systems. Testing is another problem since most layers are dependent on each other, testing a specific layer requires mock data or simulation of the related layers, which could decrease the quality of the testing.

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.

2.1.2 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 module, or multiple schemas within a single database. It is important to keep in mind that only the code base is independent in the software structure, meaning that since modular monolithics is still follows the monolithic architecture, the code will be compiled together as a single application and run in one process.

One of the benefits of using the modular monolithic structure is the domain partitioning. Similar to the layered architecture, this allows to have separate development teams that specializes in one or more of these domains, which will increase the efficiency of the work. Similar to other monolithic applications, the performance of the application will be fast since there are no network calls within the application. Furthermore, since all modules are separated from each other, it is easier to maintain the code.

However, code could be hard to reuse since each module have its own code. The logic and utilities could be tightly coupled to each domain, decreasing the flexibility to reuse the code compared to the modular monolithic structure. Also, similar to other modular monolithic architecture applications, as the applications grows, the logics and the databases could become complicated since all the code base is in one single application.

2.2 Distributed Systems

Distributed architecture is a design approach where the logical components of an application are divided 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 service, which could be a huge benefit.

There are multiple advantages to choosing distributed systems. The one that contrasts with the monolithic architecture is scalability. Since distributed architectures deploy different logical components separately from one another, it is easy to add new services. Furthermore, similar to modular monolithics, distributed architecture encourages a high degree of modularity. This is because distributed systems have loosely coupled logical components. The benefit of having multiple distributed systems is also shown during the testing phase as well. Since each unit is loosely coupled, they could be tested separately. This becomes a huge advantage as the size of the application grows compared to monolithic systems. Lastly, fault tolerance is a big difference compared to the monolithic system. Since multiple units are deployed in several units, even though one part of the system is down, the other parts of the system could still be available, depending on the coupling.

On the other hand, there are also disadvantages when choosing to build a distributed systems architecture. Performance is generally slower than a monolithic application since distributed system 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. Another aspect is cost, since 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. This leads to more complexity in the system, and makes the developers harder to understand the overall design. Also, 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 advantages and disadvantages of 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, then we could decide on which subcategory of either of the architectures to implement. Same as the monolithic structure, distributed systems architecture has multiple types of subcategories, each having advantages and disadvantages.

2.2.1 Microservices Architecture

A microservice is a service that is a separately deployed unit of software that performs some business or infrastructure process [5]. Microservices architecture is part of the layered system, where microservices communicate with each other to make an application. Since the system is divided into multiple parts, it is essential when making decisions how to divide the application, such as deciding how small or how big each microservices could get. Generally, it is better to make microservices smaller for multiple reasons. Some of the factors that are 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 parts in a separate microservice will decrease the probability of shutting down the whole system. For security and authentication, it is important to have these access control in 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 having 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 this 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. Furthermore, if a single request requires separate microservices to communicate with each other, this request is coupled and recommended to kept in the same service.

Workflows

After setting up multiple microservices, the next question is how these microservices communicate with each other to make the whole application. This is a crucial part of the microservices architecture, and this is where workflow comes in. The term workflow means when two or more microservices are called for a single request. Workflows are in charge of navigating which microservice to start, which one to call next, and which one to end with. There are two ways to handle workflows: centralized workflow management and decentralized workflow management.

The first type of workflow management style is the centralized 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 handle errors.

The advantage of this workflow management style is that the order of microservices for each request is clear. The central microservice always has the exact routes the request has to take, which allows to track the status of where all requests stopped and where to restart. This allows to handle errors efficiently and easy to change a workflow.

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.

The next type of workflow management style is the decentralized workflow. This 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 not to 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 do 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 no central microservice that has the order of microservices that is need 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.

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 the shared service are that even though a code is changed in the shared microservices, code in other microservices are not needed to be changed. 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 the 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.

A shared library is a more common way for code reuse. A library will be built, including all the code that is reused in different microservices. Once the 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 are better than shared service, since it is not remote, and the 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.

Pros and Cons of Microservice Architecture.

Microservice architectures are commonly used as they have several advantages. Loose coupling is one of them. Each microservice is single-purpose, and they are deployed separately, which makes them easy to maintain. This means that it is easy to change a particular microservice that needs modification, instead of making multiple changes in different parts of the whole application. 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 when using the microservice architecture. This architecture is complex. It requires multiple decisions, such as the aspects discussed above (workflow, shared code etc.) depending on the situation. Because of its complexity, 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.

2.2.2 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 [5] The event contains data, and it is broadcast 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 for the response of the receiving service. This is the key difference between an event and a message. Messages include a command or some kind of request for the receiving service, and the service sending out the messages requires a response, making it synchronous. Messages are also sent to only a single service using queues.

Asynchronous Communication

EDA relies on asynchronous communication when sending and receiving events. This is because events in the EDA do not wait for a response and is built for loose coupling. Asynchronous communications do not need to wait for a 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 for the response, which means that the service in response must be available. This make tighter coupling between services, making it not suitable for EDA.

Pros and Cons of EDA

Asynchronous communications have 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.

Event-Driven Architecture is highly decoupled, which makes all services independent and easy to maintain. Furthermore, 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.

On the other hand, there are also disadvantages that need to be considered. Similar to microservices, EDAs are complex. Deciding which database topology for the architecture, asynchronous communications and parallel event processing add complexity to the application. Also, asynchronous communication makes it more difficult to test out the program. Since the request is processed 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.

CHAPTER 3

Databases

3.1 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.

3.1.1 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 [11]

3.1.2 Non-Relational Databases

Nonrelational databases mean 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 nonrelational 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 on 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 also an issue since the state of the database changes over time as structures might not be consistent.

3.2 Database Design and Evolution

The design of databases and querying the database are the ones that have the most impact on the performance of the database. In this section, we will be investigating strategies to design the database to increase performance and efficency, consider scalability and evolution of databases, and ways to optimize the database and queries.

3.2.1 Normalization

Compared to a non-relational 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 several levels of normalization, and as the normalization form gets higher, it generally indicates a more refined relational design. However, most databases tend to be needed until the third normal form, which avoids most of the problems common to bad relational designs.

The First Normal Form (1NF)

The two criteria that 1NF meet are the following:

1. The data are stored in a two-dimensional table.
2. There are no repeating groups.

Here, the repeating groups mean that if an attribute has more than one value in each row of a table [7]. 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.

Table 3.1: Order table before 1NF (with repeating groups)

OrderID	**CustomerName**	**Items**	**TotalPrice**
1	Alice	Item1, Item2, Item3	$45.00
2	Bob	Item1, Item4	$30.00
3	Charlie	Item2, Item5, Item6, Item7	$70.00

This table shows the example of a table before the first normalized form. We can see that there are multiple items in the items' column separated by a comma. 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 simplest normalization that could be done in a relational database, and it is pretty clear.

Table 3.2: Order table after applying 1NF (no repeating groups)

OrderID	**CustomerName**	**Item**	**Price**
1	Alice	Item1	$15.00
1	Alice	Item2	$20.00
1	Alice	Item3	$10.00
2	Bob	Item1	$10.00
2	Bob	Item4	$20.00
3	Charlie	Item2	$15.00
3	Charlie	Item5	$10.00
3	Charlie	Item6	$25.00
3	Charlie	Item7	$20.00

As we can see on the table above, there is only one item in the items column by creating a new record for each item. However, the first normalization form is not enough to benefit 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 updating other records that are associated. Missing any of the records with the student name will result in data inconsistency. Inserting a new entity could be difficult unless there is related data. For instance, adding a new course to the table is not possible until there is a student taking the course. This is because 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 criteria that 2NF meet are the following:

1. The relation is in first normal form

2. All non-key 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 [7]. 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 on 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, the Student name and birthday would be dependent on the student id, so the student id will be the primary key. The courses the student takes do 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:

- **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:

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

Table 3.3: Student table before applying 2NF (already in 1NF, but with partial dependencies)

Student_ID	**Student_Name**	**Birthday**	**Course_ID**	**Course_Name**
S001	Alice	2002-03-14	C101	Database Systems
S002	Bob	2001-06-02	C102	Data Structures
S003	Charlie	2002-07-22	C101	Database Systems
S004	Diana	2003-01-18	C103	Operating Systems

Table 3.4: Decomposition into 2NF

Student Table			
Student_ID (PK)	**Student_Name**	**Birthday**	**Course_ID (FK)**
S001	Alice	2002-03-14	C101
S002	Bob	2001-06-02	C102
S003	Charlie	2002-07-22	C101
S004	Diana	2003-01-18	C103

Courses Table	
Course_ID (PK)	**Course_Name**
C101	Database Systems
C102	Data Structures
C103	Operating Systems

By doing this, some 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 a 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 and the table:

- **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*)

Table 3.5: Student Table (same as 2NF)

Student_ID (PK)	**Name**	**Birthday**	**Course_ID (FK)**	**Room_Name**
S001	Alice	2002-03-14	C101	Database Room
S002	Bob	2001-06-02	C102	CS Lab
S003	Charlie	2002-07-22	C101	Database Room
S004	Diana	2003-01-18	C103	Systems Lab

Table 3.6: Course Table

Course_ID (PK)	**Course_Name**	**Room_Name**
C101	Database Systems	Database Room
C102	Data Structures	CS Lab
C103	Operating Systems	Systems Lab

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 criteria 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:

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

[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).

3.2.2 Database Evolution

Database design also influences how a database evolves over time. Performance, latency, and fault tolerance are affected by the database designs as the amount of stored data increases. There are multiple techniques to design a database suitable for scaling. Vertical scaling refers to when the physical parts of the computer such as CPU or RAM are upgraded, so the database can handle more data and queries. Horizontal scaling is when multiple machines with independent physical parts (CPU and RAM), store the data on an array of disks that are shared among multiple other machines, which is also called nodes. One advantage of using vertical scaling is that the structure is very simple. Only one database needs to be taken care of, which could be easier compared to handling multiple databases. However,

Table 3.7: Decomposition into 3NF (removing transitive dependencies)

Student Table			
Student_ID (PK)	**Name**	**Birthday**	**Course_ID (FK)**
S001	Alice	2002-03-14	C101
S002	Bob	2001-06-02	C102
S003	Charlie	2002-07-22	C101
S004	Diana	2003-01-18	C103

Course Table		
Course_ID (PK)	**Course_Name**	**Room_Num (FK)**
C101	Database Systems	R12
C102	Data Structures	R14
C103	Operating Systems	R15

Room Table	
Room_Num (PK)	**Room_Name**
R12	Database Room
R14	CS Lab
R15	Systems Lab

there are more disadvantages. Cost increase faster than linearly as higher performance is required. Fault tolerance is problematic as there is only one database running, hardware or software failures on the database can cause service interruptions. For horizontal scaling, there are mainly two strategies that we will discuss: replication and partitioning. These two strategies have its own sub strategies which will be discussed in the next subsection.

3.2.2.1 Replication

Replication is when there are multiple copies of the database in different nodes, which could be in different virtual, or physical machines. This leads to redundancy in data, but it also allows for fault tolerance if one of the database hosts is unavailable [9]. Another reason to use replication is to increase availability by having databases in different regions to decrease latency. Having multiple database to read queries also help increase the performance of the application as the size of the database grows.

There are multiple ways to implement this strategy in different situations. The two topics discussed will be Leader and Followers and Multi Leader Replication.

Leaders and Followers

Going deeper into how the replicas are actually set up, it is important to make sure that each of the replicas have the same data. For example when a user updates a table, this should include all the other replicas to update the same table. Here is an example how the leader follower replication setup could look like:

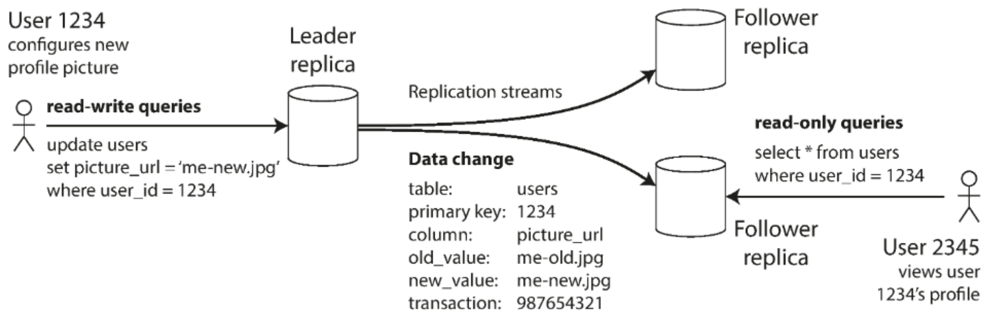

Figure 3.1: Figure of Leader and Follower replication strategy to explain the flow or write or read database querie [9].

The leader and follower structure has one lead node(leader) that allows writes to the database, and the rest of the nodes are assigned as followers. If there is a change data in the leader, the followers are updated accordingly. Reading the database could be done by any of the nodes including the leader node, but writing the database could only be done by through the leader node.

Multi-Leader Replication

One downside of a Leader and Follower replication was that write queries were only allowed in the leader replica. If the application could not connect to the leader for any reason such as network causes, write queries could not be performed. Multi-Leader replication allows multiple leaders, which means that write queries could be done in multiple replicas. Although it is rare for a multi-leader replication structure to over weigh the benefits among the complexity added, there are a few use cases for this structure.

- Multi datacenter operation
- Clients with offline operation
- Collaborative editing

However the biggest issue in the multi-leader replication structure is handling multiple write query conflict. For instance, if a same record was modified by multiple users on different leader replicas, there will be different data among replicas resulting in data inconsistency. This was not an issue for a single leader and followers structure since only one leader accepted write queries, and they were processed one by one. However, if multiple leaders receive queries at the same time, the conflict should be resolved. There are multiple ways to handle write conflicts.

The simplest strategy is to just avoid the conflicts. If the write queries could be avoided in the application layer, where the application ensures that all write queries for a particular record go through the same leader. This would work until a particular records needs to change its designated leader. In this case, simply avoiding conflicts in the application layer would not work

The next strategy is to converge toward a consistent state. When there is a situation where avoiding conflicts do not work, the conflicts should be handled. There are various ways to converge the replicas into a consistent state. One way could be to simply give each write query a unique ID, and if there is a conflict, the higher unique ID overwrites the conflict. Another way is to give each replica a unique ID, and the higher unique ID overwrites the conflict. Or, there could be another explicit data structure that preserves the information and write application code to resolve the conflict in a certain logic. There are trade-offs such as which part of the data is going to be saved over the other, so it is essential to choose the write way to handle write conflicts in a multi-leader replica structure.

Communication between replicas: Synchronous vs Asynchronous Replication

Communication that happens between the leaders and the followers is also another important factor to decide whether it is a leader and follower setup or multi-leader setup. The two types of communication are synchronous and asynchronous. After the leader receives a write query, synchronous replication waits until all the followers are updated and synced. The leader will not take any queries before this process is finished. However, asynchronous replication keeps taking queries regardless of the status of the followers. Although handling requests to copy status of the leader on the database replicas are fast, which usually happens less than a second [9], there is no guarantee of how long it would take depending on the physical status of the replica and the network. This is an advantage of synchronous replication, since it is certain that all database replications are up-to-date with the leader database, but it could be a disadvantage in a perspective where the database queries will be slow. If all of the followers are synchronous, the system would be too slow. Most of the times, one of the followers will be set to being synchronous so that there will be at least two replications(one leader, one follower) up-to-date with all the write queries, and the rest of the followers being asynchronous. This configuration is sometimes also called semi-synchronous.(Designing Data-Intensive App ch5)

Asynchronous replications on the other hand are very fast since the leader does not have to wait until the followers have synced data. However, it does have a disadvantage that all of the followers might not be up-to-date right away, which is also known as the replication lag. This might sound like a huge disadvantage, but asynchronous replication is often times used if there are many followers or if they are geographically distributed. Eventually, all the followers will have a synced replication of the leader, since inconsistency is just a temporary state. This effect is called as eventual consistency [9]. If the application that is being created is okay with possibility of inconsistency due to replication lag, asynchronous replication is perfect. However, if consistency is crucial to the application, other solutions such as synchronous or semi-synchronous replication will be the go-to when using leaders and followers method.

3.2.2.2 Partitioning

Partitioning is when a data is divided into multiple smaller pieces of data such as a record, a row, or a document. Replication was used for scalability and performance, but as the size of the database increases, only replication itself become inefficient since all the data has to be copied to the replicas. In order to scale database applications, partitioning is commonly used. As the data is distributed among multiple partitions, multiple queries could be run at the same time.

Partitioning by Key-Value Data

Having a key-value data is one way to create partitions. The reason for partitioning was to scale the database by distributing the query load on one node. One way to achieve this is to have a key range for each partition. For example, when we think of a dictionary, we know that words that start with the letter “a” is in the beginning, the letter “m” is somewhere in the middle, and the letter “z” is at the end. By dividing the range of partitions by the key-value data, it will be easier to locate which partition contains the data and distribute the query. In this case, the partitions might not be evenly divided since each range of key-value pairs could contain a different amount of data. It is important which key-value pair to use. If a key is assigned as a column that is not evenly distributed, this leads to a hot spot (a partition with disproportionately high load) [9].

A method that has a lower chance of being affected by hot spots is partitioning by hash of key. A good hash function makes the key-value pairs evenly distributed among the partitions. However, this looses the efficiency that the key range had. It is now hard to find the range queries efficiently without the key range.

When choosing the strategy, hashing keys and having key ranges to partition data both have advantages and trade-offs, so it is important to choose the one suitable in the situation. Key range could be used if there is a key range that is evenly distributed, and key hash when there are still a lot of hotspots by using the key range.

Partitioning and Secondary Indexes

Partitioning by key-value pairs work nicely with the partitioning strategies discussed above. However, things get more complicated with secondary indexes. Secondary indexes will be discussed more in the upcoming sections. A brief introduction for secondary indexes is that they are created to improve the query performance that are based on multiple keys. For example, when searching for a job in a job board, selecting criteria such as industry or experience level could be a situation to use secondary indexes. Secondary indexes make sharding more complicated since they don’t identify a single record, but rather search for records based on the conditions. Partitioning by key-value pairs wouldn’t work in this situation.

Scatter/gather is a technique where each partition has its own secondary indexes. In this case, writing data would be efficient since only the partition containing that specific part of the index is where the data is written. However, reading queries are less efficient. If queries require searching on different partitions, each partition is searched separately which is inefficient.

It is also possible to partition secondary indexes by the term. Instead of having distributed secondary indexes across the partitions, this approach has a global index that covers all the partitions. Having a range of secondary indexes partitioned in separate partitions make read queries more efficient from the scatter/gather technique. If write queries are used more often, it is better to use the scatter/gather technique, and if read queries are dominantly used, partitioning indexes by term is the structure to choose.

Request Routing

After partitioning the dataset into multiple nodes, it is important to make sure the request connects to the right partition. On a high level there are mainly three ways to achieve this.

1. Allow clients to contact any node (e.g., via a round-robin load balancer). If that node coincidentally owns the partition to which the request applies, it can handle the request directly; otherwise, it forwards the request to the appropriate node, receives the reply, and passes the reply along to the client.

2. Send all requests from clients to a routing tier first, which determines the node that should handle each request and forwards it accordingly. This routing tier does not itself handle any requests; it only acts as a partition-aware load balancer.

3. Require that clients be aware of the partitioning and the assignment of partitions to nodes. In this case, a client can connect directly to the appropriate node, without any intermediary [9].

The key problem is that how does the component, (which could be the routing tier, the client, or the partition in the node) know about changes in the other partitions in other nodes? If the deciding component is not up-to-date with the location where data that the client is requesting, this could lead into data inconsistency.

To solve this problem, many distributed data systems use a separate coordination service such as the ZooKeeper to keep track of the data. The whenever a partition changes or a node is added or removed the ZooKeeper is updated to keep data consistency.

3.3 Optimization

Several factors influence the performance of the database. The design of the database, including normalization and partitioning, which was discussed in the previous subsection, is one of the factors. Indexing is another huge part of increasing the performance of the database as the amount of data increases. Lastly, changing the physical query of the database is a way to optimize the performance of the database. In this section, we will be discussing specific ways to optimize the performance of databases.

3.3.1 Indexing

Suppose that there is a table called students with the following keys: id, name, grade. Here is an example table:

ID(PK)	**Name**	**Grade**
1	Alice	90
2	Bob	80
3	Charlie	70
4	David	70
5	Eva	80

Table 3.8: Original Students Table with ID, Name, and Grade

If we want to find students with a certain grade (80%), we could run a SQL query something like this.

```
SELECT *
FROM Students
WHERE grade = 80;
```

This query will visit all the rows and then find the students with grades 80. However, this query becomes more inefficient as the size of the table grows, as the time complexity would be O(n). If the table was sorted by grades, this will be much easier, since we could do a binary search to make the search speed O(log n). This is where indexing is applied.

An index could be created where the grades are sorted in order, each having a reference to the row of the student table. This way, whenever we want to find the name of the students with a certain grade, instead of looking at the Students table, we can look at the index, where the grades are sorted, find rows with a certain grade, for each row, find the reference of the student table, and return the name of the student. Here is an example how the index might look like:

Grade	ID
70	3
70	4
80	2
80	5
90	1

Table 3.9: Index on Grade: sorted by Grade with references to Original Table

In this index, grades are sorted by order, and they include a reference to the student table's primary key, which is the student's id. This will significantly improve the read queries for databases for large sets of data.

However, indexes take up additional memory and can slow down write operations, such as inserts, updates, or deletes. It is important to use indexes on every column. Indexes should be used in situations where they could significantly increase the performance of read queries.

There are multiple data structures that could be used to implement indexes, but the most common ones are b+trees. B-trees are somewhat similar to the binary search trees. However, instead of having one value for every node, it contains multiple values for each node.

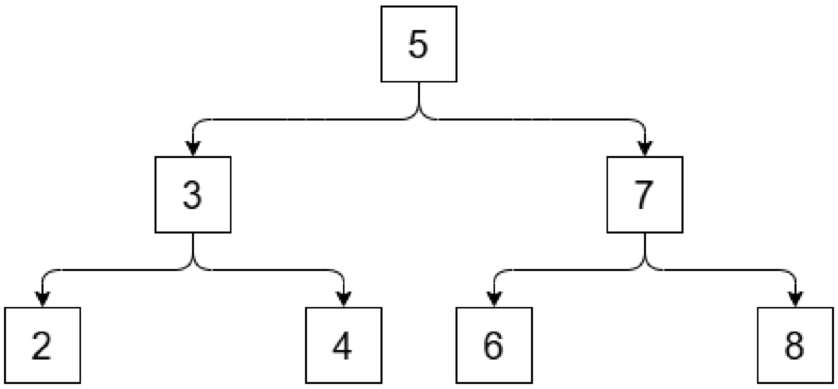

Figure 3.2: Model of a binary tree to explain datastructures used for database indexing.

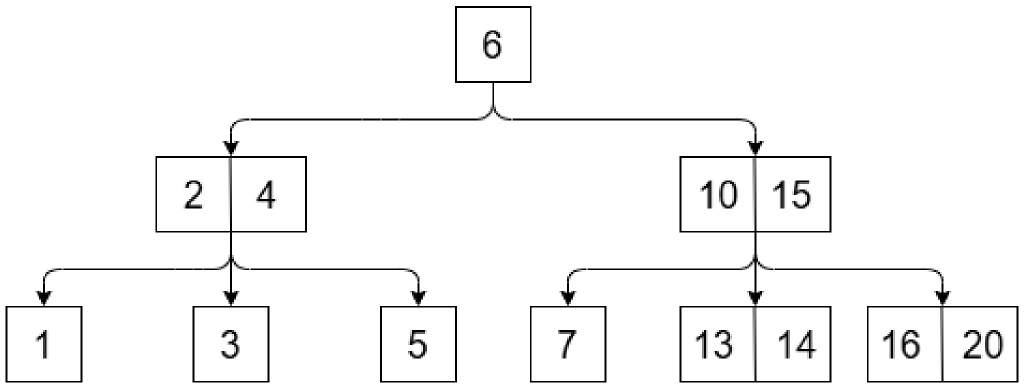

Figure 3.3: Model of a b tree to explain datastructures used for database indexing.

As we can see from the figures above, b-trees can hold multiple nodes, such as nodes 2 and 4, or 10 and 15. This allows multiple partitions, allowing a balanced and faster search. Furthermore, as multiple values could be in a single node, the height of the tree would be smaller. This means that there are fewer disk I/O's per file operation as the database stores data on disk, resulting in faster query execution [6]. One down side of b-trees is when making range queries, such as finding all students who have grades 20 to 80. If these two data are separated from the root, this could be inefficient. B+trees are a variation of b-trees to solve this issue.

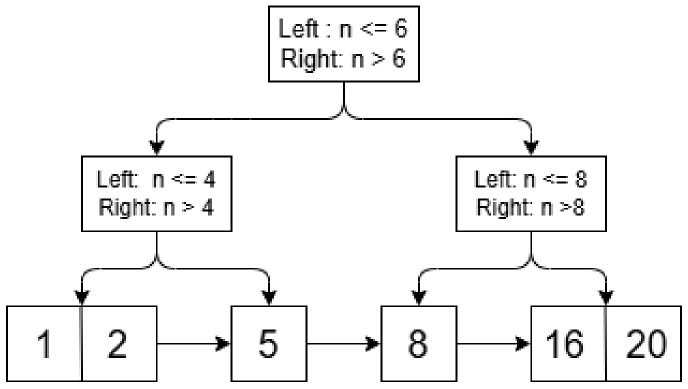

Figure 3.4: Model of a b+ tree to explain datastructures used for database indexing.

B+trees store data only in the leaf node. The other nodes store keys only for navigation, like the explanations on the internal node (less than equal to six goes left, greater than six goes right) on the figure above. Also, all the leaf nodes are connected to the next leaf node. B+trees are commonly used in indexes for modern database management systems such as MYSQL, PostgreSQL, and SQLite.

3.3.2 Query Optimization

Optimizing the SQL queries could also increase the performance of the database. Here are the 6 queries tune SQL queries. https://www.geeksforgeeks.org/sql/sql-performance-tuning/

- SELECT fields instead of using SELECT *
- Avoid SELECT DISTINCT
- Use INNER JOIN instead of WHERE for Joins
- Use WHERE instead of HAVING
- Limit Wildcards to the end of a search term
- Use LIMIT for sampling query results

3.3.3 N+1 problem

The N+1 problem is a common problem when using the ORM. The definition of the problem is when an application retrieves a list, then performs additional queries for each item's related data, which results in an inefficient query. (1 + N queries instead of optimized joins). In a nutshell, it is just having too much queries that decreases the performance. This problem usually happens in ORMs.

Similar to the example above, assume there is a students table with keys student_id, student_name, and another table called courses with the keys course_id, student_id, and course_name.

If we were to get all students and courses they are taking in plain SQL, we would write something like this:

```
SELECT s.student_id, s.student_name, c.course_name
FROM students s
JOIN courses c ON s.student_id = c.student_id;
```

Using an ORM, we could do something like this:

```
students = session.query(Student).all()
for s in students:
    for c in s.courses:
        print(s.name, c.name)
```

This is where the n+1 problem happens. The way of getting all students and their courses through the ORM translates to the following raw SQL query:

```
SELECT * FROM students;

SELECT * FROM courses WHERE student_id = 1;

SELECT * FROM courses WHERE student_id = 2;

SELECT * FROM courses WHERE student_id = 3;
```

This is inefficient since we are performing 1 query (`SELECT * FROM students`), and then performing *n* queries (`SELECT * FROM course WHERE student_id = n;`). The performance of the database will get worse as the size of n increases. It might be questioned why the ORM translates to inefficient SQL code, and the reason for this is lazy loading. Many ORM frameworks have lazy loading as the default, and only call the related data when it is accessed. In this example, the ORM does not know that all the students and their courses are going to be fetched. They only understand each iteration of the nested for loop as a separate query and fetch the data for that specific iteration [8].

There are a few ways to solve the n+1 problem, depending on the tools available by the ORM. Most ORMs provide eager loading, which allows for fetching the related data right away instead of waiting until specific data is called. This is the opposite of eager loading. This reduces the need for additional queries and increases the efficiency when fetching data. Most ORMs provide customization to decide when and how to perform eager loading, allowing optimization when querying databases. For example, if we want to use eager loading in SQL Alchemy, we would do something like this:

```
from sqlalchemy.orm import joinedload


students = session.query(Student).options(joinedload(Student.courses)).all()
for s in students:
    for c in s.courses:
        print(s.name, c.name)
```

The difference from the previous query is that instead of just fetching data from the students table, this query fetches all course data related to the student beforehand to prevent lazy loading.

Another way to solve the N+1 problem is batch fetching. Batch fetching still keeps the idea of lazy loading, but instead of fetching a single row of data, it fetches multiple rows of data at once. For example, we can write something like this in SQL Alchemy:

```
from sqlalchemy.orm import relationship


class Student(Base):
    __tablename__ = ”students”
    id = Column(Integer, primary_key=True)
    name = Column(String)
    courses = relationship(
        ”Course”,
        back_populates=”student”,
        lazy=”selectin”,       # enable batch fetching
        batch_size=10          # load 10 parent IDs per batch
    )

students = session.query(Student).all()
```

When creating the relation in SQLAlchemy, we can specify the batch size. In this example, by setting `lazy=”selectin”` and `batch_size=10`, we are telling the ORM when to fetch courses from students and to load 10 student IDs per batch, resulting in the following query:

```
SELECT courses.id, courses.name, courses.student_id
FROM courses
WHERE courses.student_id IN (1,2,3,4,5,6,7,8,9,10);

-- Batch 2
SELECT courses.id, courses.name, courses.student_id
FROM courses
WHERE courses.student_id IN (11,12,13,14,15,16,17,18,19,20);
```

This will avoid the N+1 problem by instead of fetching for n queries, it will fetch for m queries, where m is n/batch size. Furthermore, batch fetching is useful when the size of the data becomes larger. Unlike eager loading, this avoids data duplication, where we do not get the same student row repeated for every course. Being able to adjust the batch size is also beneficial when scaling the application.

CHAPTER 4

Caching

4.1 What is Caching

Caching data means to store frequently accessed data in a location that is easy and fast to access (usually in RAM). The reason why caching is used is to ultimately increase the performance and efficiency when retrieving data. Data in the database are usually stored in a disk, which could be a hard drive or an SSD, which usually takes more time to process I/O. Caching could help decrease the amount of time to process this data significantly. One might suggest then why not put all the data in the cache, but this would not work since the cache could not store a lot of data as much as the database does. The cost of the hardware of the cache is much more expensive, and as the size of the cache increases, the search time will also increase, decreasing the speed and defeating the whole purpose of caching. Caching can be used at several levels. This article will mainly focus on caching in databases and web pages. There are multiple strategies that could be used when caching these two areas. Another important part of caching is the invalidation strategies. Deciding when the cache expires and when it updates could affect the performance and data consistency. Choosing the right technique in the given situation will be essential.

4.2 Database Caching Strategies

There are multiple database caching strategies that could be applied when designing cache systems. Depending on the type of request (read, write) of the user, and which part of the application is responsible for fetching data from the database or managing the cache, there are five main database caching strategies.

Cache Aside

Cache Aside is also called Lazy Loading. The application is in control of managing the cache. Let's look at the diagram below.

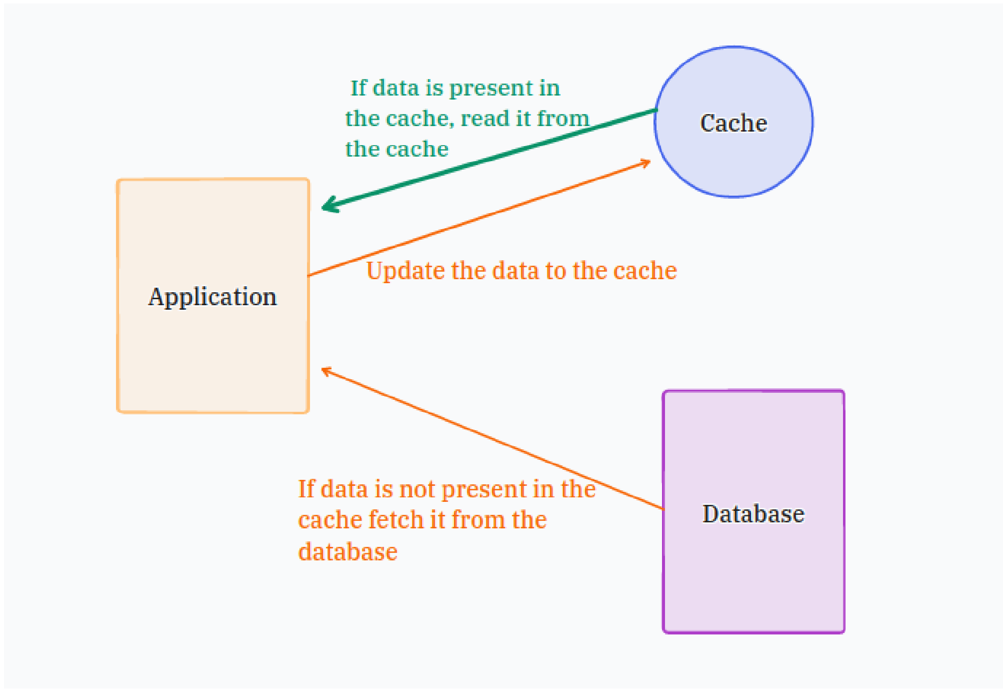
If data is present in the cache, read it from the cache
Cache
Application
Update the data to the cache
If data is not present in the cache fetch it from the database
Database

Figure 4.1: Model of Cache-aside Pattern to explain common database caching strategies [12].

As the diagram shown above, the client first checks the cache to see if the read request data is in the cache. If the cache exists, this is called a cache hit, and the client uses it right away. If the cache does not exist, this is called a cache miss. When a cache miss occurs, the client then sends a request to the database server to fetch the data. After that, the client transfers the data to the cache, so that there could be a cache hit the next time the data is needed. This will increase the performance of the application when the same data is called multiple times. When there is a write request, the application will first communicate with the database and then update the cache.

This strategy is useful when the application requires a lot of read requests from the database, since the application could just check the cache instead of querying the database. One disadvantage of this approach is that there could be an inconsistency between the cache and the database. Data directly written in the database might not be consistent with the data in the cache, since writing data to the database and writing data to the cache does not happen at the same time.

Read-Through

In the read-through strategy, the cache level manages fetching data from the database. Here is a diagram of how the read-through strategy works:

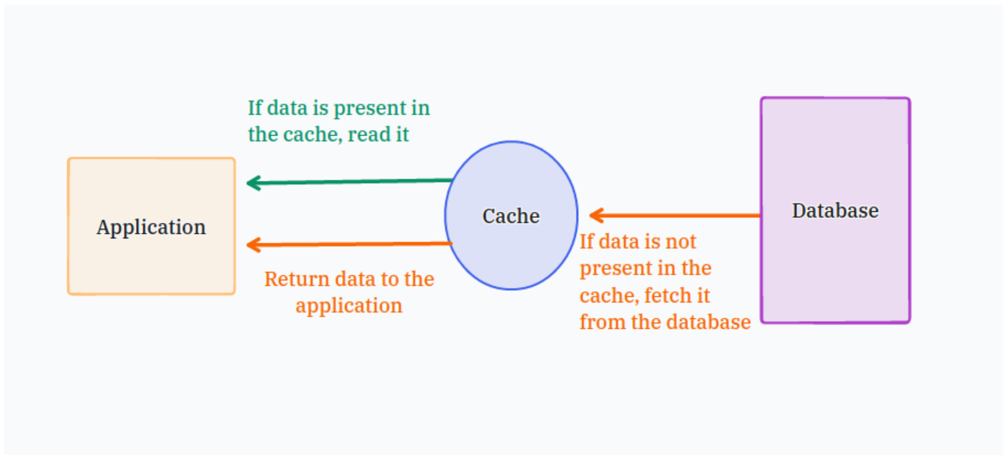
If data is present in the cache, read it
Application
Cache
Database
Return data to the application
If data is not present in the cache, fetch it from the database

Figure 4.2: Model of Read-Through Pattern to explain common database caching strategies [12].

Whenever there is a read request from the application, the application first checks the cache. If there is a cache hit, it simply returns the data back to the application. If there is a cache miss, the cache level fetches the data from the database. Since the cache manages fetching data, it simplifies the application logic when retrieving data compared to when the application handles fetching data.

This strategy only involves read requests from the application, which means that other strategies could be used for write requests. The write-through strategy is generally great to be used with the read-through strategy.

Write-Through

The write-through strategy is very similar to the read-through strategy. The diagram is almost identical.

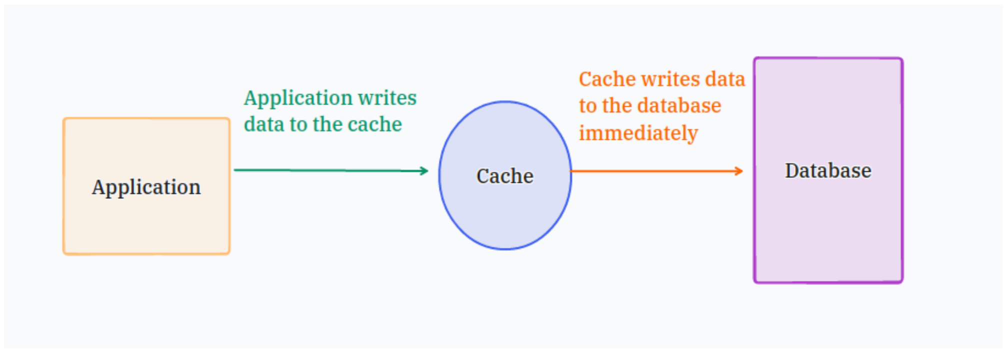
Application writes data to the cache
Cache writes data to the database immediately
Application
Cache
Database

Figure 4.3: Model of Write-Through Pattern to explain common database caching strategies [12].

Nothing happens when there is a read request from the application, but when there is a write request, the data will first be written in the cache and then into the database. This ensures data consistency when paired with the read-through strategy. One downside of this strategy is that since it writes to both the cache and the database layer, the latency of the request increases.

The read-through and the write-through strategies are not meant to be used for all access to data in the application layer. Using these strategies for all database queries could result in a decrease in performance, since the caching layer is not intended to keep every single data.

Write-Back

The write-back strategy is similar to the write-through strategy, but the writes to the database happen with a delay. The cache is still in charge of writing data to the database. Let's look at the diagram:

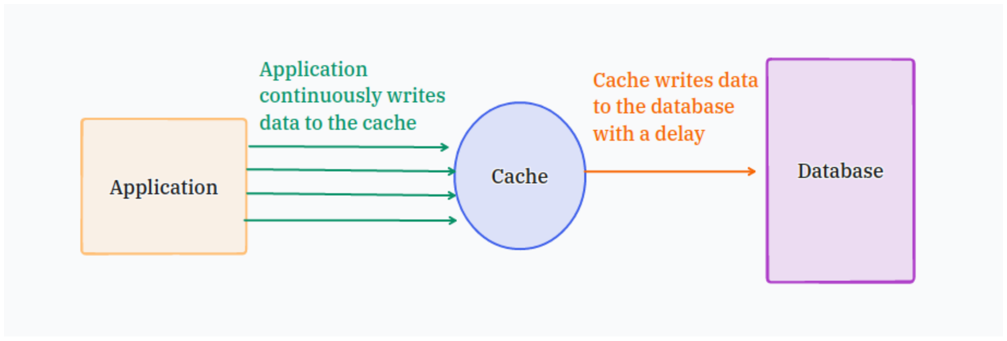

Figure 4.4: Model of Write-Back Pattern to explain common database caching strategies [12].

Since the cache only writes to the database after a certain amount of time, this strategy is beneficial when there are multiple write requests to the same data at the application level. This will decrease the write queries from the cache level to the database, which could increase the performance. However, similar to the read-through and write-through strategies, since the data is written to the cache first, writing data to the database could potentially fail if the caching fails.

Write Around

This strategy is similar to the cache-aside strategy, but it has more specific instructions for write requests.

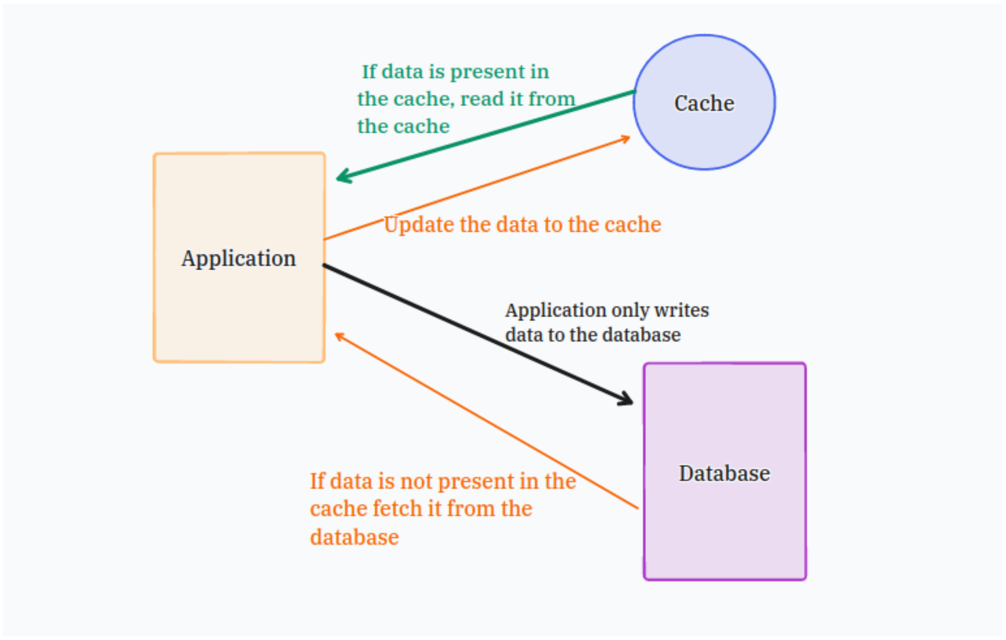
If data is present in the cache, read it from the cache
Cache
Update the data to the cache
Application
Application only writes data to the database
Database
If data is not present in the cache fetch it from the database

Figure 4.5: Model of Write-Around Pattern to explain common database caching strategies [12].

The difference between the write-around strategy and the cache-aside strategy is that when there is a write request, only the data in the database is updated. The data in the cache is only updated when there is a cache miss from a read request. This way, the cache is not overflooded with unnecessary data. The disadvantage of this strategy is that recently written data will always result in a cache miss. This is why the write-around strategy is suitable when the application has a lot of read requests and data is rarely updated.

4.3 Redis

Until now, we have learned the types of database caching strategies. But how are these strategies actually implemented? Redis is a widely used open-source data structure store that makes these caching patterns practical at scale.

Redis works entirely in memory, which gives it extremely fast read and write speeds. Most databases store data on disk, so every query involves slower I/O operations. Redis avoids this by keeping data in RAM, allowing applications to access cached values in microseconds. This speed makes it perfect for handling database caching strategies [4].

Redis also supports a wide range of data structures—strings, hashes, lists, sets, sorted sets, and more. These structures map naturally to different caching scenarios. For example, strings work well for caching simple query results, hashes can store user profile objects, and sorted sets can maintain real-time rankings or leaderboards. Because Redis supports many data structures, it is not limited to simple key–value caching. It can also function as a fast and lightweight data engine that handles situations that require complex caching.

Reliability is another reason Redis is chosen for database caching. Redis provides persistence options (RDB snapshots and AOF logs) so cached data or application-critical information can survive server restarts. It also supports replication and clustering, which distribute data across multiple nodes to achieve both high availability and horizontal scalability. These features ensure that the caching layer does not become a single point of failure.

In summary, Redis transforms abstract caching strategies into practical and high-performance solutions. Its low latency, adaptable data model, and robust operational features make it a dependable component in web applications. These are the reasons why Redis will be used to implement database caching strategies in this project.

4.4 Client Side Caching

People find it annoying when it takes too long or laggy to load a website they want to access. This is where client side caching could be beneficial. Client side caching is a technique where data is cached inside the client's device unlike the application level caching. Frequently used static data are normally cached on the client side to improve the performance and efficiency of web applications. The reduced time to load the webpages result in enhanced user's satisfaction and the overall experience. Static data could also be available when the user's device is offline, which could be useful for some mobile and web applications.

There are different types of client side caching that could be used, but the most common ones are local storage, session storage, and browser cache. Local storage and session storage are types of web storage APIs, where developers can store data on the client side that is not automatically cached by the browser. This could include saving users' preferences such as theme or language, or simple offline data. The difference between local storage and Session storage is the scope of the browser. Local storage is shared among all tabs and windows of the same origin, but session storage is specific to a single tab. For session storage, they will be deleted once a tab is closed. However, local storage will be saved even after the tab and the browser are closed.

Browser cache is where most of the static elements of the web application, such as HTML, CSS, JavaScript, and images, are automatically cached. The way how and where these caches are stored depend on the browser the user is using, but the core concepts are the same. Although browser cache is automatically stored, there are some parts that could be controlled within the browser cache, which is called cache control. Cache control is an HTTP response header that gives instructions to the browser on how to cache resources. There are 15 total cache control techniques, and each of these techniques serves different purposes, depending on how often the data changes and how much freshness the application requires, but we will be discussing the most common ones.

4.5 Caching Invalidation

After a cache is stored whether in the client or the server side, it is crucial to know when the cache should be validated. For example, let's assume the user's profile is cached in the browser. Without any caching invalidation, even after the user's profile is updated, the user will not be able to see the changes if the cache still exists in the browser. This could lead to crucial data inconsistency as the importance of the data increases such as sensitive security data. Caching invalidation strategies are needed to set when the cache should be stored, updated, or deleted. Choosing the right caching invalidation strategy can lead to better performance of the application, freeing up the memory space and increasing cache-hit rates.

One of the most common types of cache invalidation is time-based invalidation. This type of technique invalidates cached data after a certain time period has passed, which could be decided by the developers [10]. Although it is simple and widely used, it also has limitations because it may invalidate data too early or too late depending on the situation. In some systems, a timeout value can be added directly in a cache configuration file to specify how long the entry should stay valid. This value can also be customized to fit the needs of the application. Some examples of time-based invalidation could be a recipe website may refresh its cache daily to show the latest trending dishes, or a movie streaming service that could be refreshed where there are new releases every few hours.

Another type of invalidation is event-based invalidation, which happens when a specific event is triggered in the system. This is useful when the cached data is tied closely to an event or state change. For example, when a blog post is updated, the old cached version of the post should be removed so that users can see the most recent version right away. This ensures that the content stays consistent with changes happening inside the system.

Another type is group-based invalidation, which removes cache based on a larger group or category. This is helpful when many cache entries belong to the same section of the application, and clearing them one by one would be inefficient [10]. This type of cache invalidation technique is commonly used when a larger group of data must be invalidated at once. For example, on a news website, if the politics section is updated, all cached articles under that section should be invalidated to make sure the latest content is displayed. In eCommerce systems, updating a product category may require invalidating all caches related to that category.

Lastly, there is validation-based invalidation. This cache invalidation technique relies on the browser checking with the server to see if the cached data is still valid instead of removing it after a fixed time or when an event occurs [10]. A common technique for this is using ETags, which are small identifiers generated by the server to represent a specific version of a resource. When the browser stores a cached file, it also keeps the ETag value and later sends it back to the server to confirm whether the resource has changed. If it has not changed, the server replies with a "Not Modified" response, which allows the browser to continue using the cached version without downloading it again. This approach helps reduce unnecessary network usage while still ensuring that users receive the most up-to-date content, making it useful for situations where data changes unpredictably.

Not all of these caching strategies could be used both in the server side and the client side. Each technique serves a different purpose, and some are only effective when controlled by the system that manages the data. Time-based invalidation is mostly handled on the server side through cache headers, while event-based and group-based invalidation occur exclusively on the server since they depend on changes happening within the application itself [10]. On the other hand, validation-based methods such as ETags rely on collaboration between the browser and the server, where the server generates validation tokens and the client uses them to check freshness. Understanding where each technique is applied helps ensure that caching remains efficient, consistent, and aligned with the needs of the application.

CHAPTER 5

Implementation

5.1 General Design Choices Made

5.1.1 Backend Design Choices

5.1.2 Frontend Design Choices

CHAPTER 6

Results

CHAPTER 7

Challenges and Lessons Learned

7.1 What worked well

7.2 What did not work well

References

[1] Kamran Ahmed. *Backend Developer Roadmap*. 2025. url: https://roadmap.sh/backend (page 1).

[2] Tom Barker. *Intelligent caching*. O'Reilly Media, Inc, 2017.

[3] Domenico Bianculli et al. "Assessing the Impact of Asynchronous Communication on Resilience and Robustness: A Comparative Study of Microservice and Monolithic Architectures". In: *Software Architecture. ECSA 2025 Tracks and Workshops*. Cham: Springer Nature Switzerland, 2026, pp. 171–186. isbn: 978-3-032-04403-7.

[4] Josiah L. Carlson. *Redis in action*. Manning, 2013 (page 32).

[5] Raju Gandhi, Mark Richards, and Neal Ford. *Head first software architecture*. O'Reilly, 2024 (pages 6, 9).

[6] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems: The complete book*. Pearson Education Limited, 2014 (page 24).

[7] Jan L. Harrington. *Relational database design and implementation*. Morgan Kaufmann/Elsevier, 2016 (pages 12–13, 16).

[8] Jacky. *Solving the notorious N+1 problem: Optimizing database queries for Java backend developers*. Sept. 2023. url: https://dev.to/jackynote/solving-the-notorious-n1-problem-optimizing-database-queries-for-java-backend-developers-2o0p (page 26).

[9] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly, 2017 (pages 17–21).

[10] Redis. *Cache invalidation*. Aug. 2025. url: https://redis.io/glossary/cache-invalidation/ (pages 34–35).

[11] Amazon Web Services. *Relational vs Nonrelational Databases - Difference Between Types of Databases - AWS*. What's the Difference Between Relational and Non-relational Databases? n.d. url: https://aws.amazon.com/compare/the-difference-between-relational-and-non-relational-databases/ (page 10).

[12] Maharani Thiraviyam. *What is caching strategies in DBMS?* July 2025. url: https://www.geeksforgeeks.org/dbms/what-is-caching-strategies-in-dbms/ (pages 29–32).

[13] Alexi Turcotte, Mark W. Aldrich, and Frank Tip. “reformulator: Automated Refactoring of the N+1 Problem in Database-Backed Applications”. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’22. Rochester, MI, USA: Association for Computing Machinery, 2023. isbn: 9781450394758. doi: `10.1145/3551349.3556911`. url: `https://doi.org/10.1145/3551349.3556911`.