Database Exam

18/12-24

Jannah Naim Ahmed Al-Kassab

jaal0004@stud.kea.dk

WEBU-GBG-W24A, Databases

Primary Keys, Foreign Keys, and Database Relationships

In a Document Database, the structure differs from traditional systems that rely on concepts such as “foreign keys.” Instead, each document is typically identified by a unique primary key. Relationships between documents are formed either by embedding related data within the document itself or by referencing the ID of another document.

Embedding proves beneficial when related data is frequently accessed together. For instance, storing a user’s addresses directly within their user document can simplify the reading process, allowing for all relevant information to be retrieved at once. However, this approach can result in large documents, and may complicate the updating process.

Another common approach is to store a reference, when managing relations between documents. For instance, when using one document’s ID as a field in another, such as the “user_id” in the order document. This method helps keep the documents smaller and more adaptable, but it does not ensure data integrity. If a referenced document is deleted, it becomes the responsibility of the application to address the consequences.

When deciding between embedding and referencing data, the choice largely depends on how you access the data and the scale of the related information. Referencing is preferable for keeping documents smaller, or when dealing with a broader range of relationships, while embedding is a smart option when the data is closely linked and frequently accessed together.

Ultimately, there is no universal solution. Document databases provide the flexibility to either embed or reference data, which can streamline various workloads. However, if the core of your queries revolves around relationships and traversals, a graph database may be the most effective tool for your needs.

CRUD Queries

CRUD stands for **Create, Read, Update, and Delete**, which are the four fundamental operations used to manage data in a database. The **Create** operation allows for the insertion of new records into collections, such as adding new books, users, authors, or borrowed book

records. For example, when a new book is added to the system, a new document is created in the **Books** collection with details like title, author, and category. The **Read** operation retrieves existing data from the database, such as when users search for books or check which books they have borrowed. This is the most frequently used operation in the system. The **Update** operation modifies existing records, such as updating a user profile, changing book details, or updating the **return_date** of a borrowed book. This ensures that the system reflects the most current information. The **Delete** operation removes records from the database, which is useful for deleting outdated or incorrect records, like removing old books from the **Books** collection or deleting users who no longer have access. Together, these CRUD operations ensure that the system can effectively manage, maintain, and display accurate data.

Constraints

The **ON DELETE CASCADE** constraint is a key feature in **foreign key relationships**. It guarantees that when a **parent record** is removed, all associated **child records** are automatically deleted as well. This constraint is commonly used to maintain **referential integrity**.

For example, if there is a **Users** table and a **Borrowed_Books** table, and **user1** is deleted from the **Users** table, all records in the **Borrowed_Books** table that reference **user1** will also be deleted. This ensures there are no "**orphaned**" **borrow records** left in the system that reference a non-existent user.

In my system, constraints are used to maintain the **integrity of key relationships**. Since the system tracks **Users**, **Books**, **Authors**, **Publishers**, and **Borrowed_Books**, constraints are necessary to ensure that these entities remain consistent.

To ensure the uniqueness of records in key tables such as **Users**, **Books**, **Authors**, **Publishers**, and **Borrowed_Books**, I would use a **Primary Key constraint** on the key identifier columns. For example, in the **Users** table, the column **user_id** would be designated as the **primary key**. This ensures that no two users can have the same **user_id**, supporting the integrity of user data.

The **Primary Key** also allows for relationships to be created with **foreign keys** in child tables, enabling constraints like **ON DELETE CASCADE**. For example, in the **Borrowed_Books** table,

the **user_id** serves as a foreign key that references the **Users** table. If a user is deleted from the **Users** table, the **ON DELETE CASCADE** constraint ensures that all borrow records linked to that user in the **Borrowed_Books** table are also removed. This prevents the system from having "orphaned" records that reference non-existent users.

This approach helps ensure **data consistency** and reduces the need for manual cleanup of child records. By using constraints like **Primary Key**, **Foreign Key**, and **ON DELETE CASCADE**, I maintain **referential integrity** between parent-child relationships in my system.

Triggers

Triggers are **automated actions** that run in response to specific database events like **INSERT, UPDATE, or DELETE**. They are used to **maintain data consistency**, **enforce business rules**, and **automate repetitive tasks**. Unlike stored procedures, triggers activate automatically whenever the specified event occurs on a table or collection.

One major benefit of triggers is their ability to **ensure data integrity**. For example, when a parent record is deleted, a trigger can automatically delete related child records, preventing "orphaned" records. Triggers can also **track changes** in data, making them useful for audit logs and system monitoring.

In my system, I would use triggers to **automate book borrowing and returning processes**. One trigger would automatically **update the borrow count** for a book whenever a new entry is added to the **Borrowed_Books** table. Instead of manually updating the borrow count, the trigger would increase the **borrow_count** by 1 each time a user borrows a book. This ensures that borrowing statistics are always accurate.

Another trigger would **update the status of a book** from **"borrowed"** to **"available"** when a user returns a book. This trigger would be activated after the **return_date** is updated in the **Borrowed_Books** table. By automating this process, the system always reflects accurate availability of books, ensuring users see up-to-date information.

Triggers provide several benefits for my system. They **reduce manual updates**, **ensure real-time accuracy**, and **prevent errors**. By using triggers to track borrow counts and book statuses, I maintain consistency across the system without relying on manual intervention. This makes the system more efficient, secure, and easier to maintain.

Views

Views are **virtual tables** that display data from one or more tables based on a predefined query. Unlike regular tables, views do not store data themselves; instead, they present the result of a query as if it were a table. Views are useful for **simplifying complex queries**, **enhancing security**, and **organizing data** for reporting or analytics. By using views, I can avoid writing repetitive queries and create a simple, reusable way to access important information.

One major benefit of views is that they **hide the complexity of joins and filters**. For example, instead of writing a query to join **Users**, **Books**, and **Borrowed_Books** every time I want to see which books a user has borrowed, I could create a view called **User_Borrowed_Books_View**. This view would show **user names**, **book titles**, and **borrow dates** all in one place, and I could query it directly without having to repeat the joins in every query.

Views also provide **security benefits**. If I want to give users access to certain information (like the books they have borrowed) but not expose sensitive details (like user emails or passwords), I can create a view that only shows **non-sensitive columns**. This way, users can query the view without accessing the underlying tables, keeping the database more secure.

In summary, views are a simple way to **organize, secure, and simplify access to data**. They reduce the need for repetitive queries, speed up reporting, and help maintain data security by controlling what data is visible to users. This makes the system more efficient and easier to maintain.

JOIN Statements

Join statements are used to combine data from multiple tables by establishing a relationship between them. The four most common types of joins are **INNER JOIN**, **LEFT JOIN**, **RIGHT JOIN**, and **FULL JOIN**, each serving a specific purpose in retrieving and displaying related data.

An **INNER JOIN** returns only the records that have matching values in both tables. In my system, if I want to see a list of users and the books they have borrowed, I would use an INNER JOIN between the **Users** table and the **Borrowed_Books** table. This join ensures that only users who have actually borrowed books are displayed.

A **LEFT JOIN** returns all records from the **left table** (Users), even if there is no matching record in the **right table** (Borrowed_Books). This is useful if I want to see a list of all users, regardless of whether they have borrowed books. For users who have not borrowed any books, their book details would be displayed as **NULL**, but their user information would still be shown. This is helpful for identifying **inactive users** in the system.

A **RIGHT JOIN** returns all records from the **right table** (Books), even if there is no matching record in the **left table** (Borrowed_Books). This is useful if I want to see a list of all books, regardless of whether they have been borrowed. For books that have never been borrowed, their **borrow date** and **user information** will be **NULL**, but the book details will still be displayed. This join is helpful for identifying **unborrowed or unpopular books** in the system.

A **FULL JOIN** returns all records from both tables, regardless of whether a match exists. If I want to see a list of all users and all books, even if there is no borrowing relationship between them, I would use a FULL JOIN between the **Users** table and the **Borrowed_Books** table. This allows me to see users who have not borrowed books, as well as books that have never been borrowed.

SQL Queries (UNION, HAVING, GROUP BY)

SQL commands such as UNION, HAVING, and GROUP BY are essential for querying, organizing, and analyzing data within relational databases.

UNION:

The UNION command allows you to merge the result sets of two or more SELECT queries into a single cohesive list. By default, it eliminates duplicate entries; however, if you wish to preserve duplicates, you can utilize UNION ALL. This command is particularly helpful when I want to consolidate multiple queries into one unified collection.

In my system, I have **Books** stored in a table, but I also have a **Borrowed_Books** table that tracks which users borrowed which books. If I want to create a unified list of all books in the system, whether they are currently borrowed or available, I would use **UNION** to combine books that are listed in the **Books** table with books that are currently being borrowed. This

approach would be useful, for example, if I wanted to generate a list of all books for an admin report, regardless of whether they are available or on loan.

HAVING:

The **HAVING** clause is used to filter groups of data after an aggregation (like **COUNT()**, **SUM()**, or **AVG()**) has been applied. It is different from **WHERE**, which filters individual rows, while **HAVING** filters aggregated data after grouping.

In my system, I would use **HAVING** to filter authors based on how many books they have written. For example, if I want to generate a report showing only authors who have written more than 5 books, I would group the books by **author_id** and then use **HAVING** to filter out authors with fewer than 5 books. This is useful in situations where I need to display "top authors" on an admin dashboard or in an analytics report.

GROUP BY:

The **GROUP BY** clause groups rows with the same values into summary rows. It is typically used with aggregate functions like **COUNT()**, **SUM()**, **AVG()**, **MIN()**, and **MAX()** to create grouped summaries of data.

In my system, I would use **GROUP BY** to track how many books each user has borrowed. This would allow me to generate a report for the admin to show which users are the most active borrowers. To do this, I would group the **Borrowed_Books** table by **user_id** and count the number of books each user has borrowed.

Stored Procedures, Functions, and Routines

Stored procedures, functions, and routines are essential tools for automating tasks, maintaining data consistency, and improving performance within a relational or document database. A **stored procedure** is a set of SQL statements stored directly in the database, allowing it to be reused multiple times. It can accept input parameters, execute logic, and return results. A **function** is similar to a stored procedure but is typically used to return a value, and it can be used as part of a query. **Routines** is a general term that refers to both stored procedures and functions as reusable blocks of logic in the database.

Using stored procedures, I can **reduce redundancy** in my system, as I no longer need to repeat the same logic in multiple places. They also allow for better **code maintainability** because if the business logic changes, I only need to update the procedure, not every instance where that logic is used. Stored procedures run directly in the database engine, which makes them faster than executing queries from an external application.

In my system, I would create stored procedures for essential tasks such as **borrowing a book** and **returning a book**. The **BorrowBook** procedure would take inputs such as **user_id**, **book_id**, and **borrow_date**. It would check if the book is available, insert a new record into the **Borrowed_Books** table, and update the **borrow_count** in the **Books** table. This approach ensures that the borrow process is consistent and efficient while reducing errors caused by manual queries. Similarly, the **ReturnBook** procedure would handle the return of a borrowed book. It would check if a borrow record exists, update the **return_date** for that borrow record, and reduce the **borrow_count** in the **Books** table. Both procedures help maintain **data integrity** by ensuring that borrowing and returning books follow a set process with built-in checks.

Stored procedures also support **error handling**, which allows me to provide custom error messages. For example, if a user tries to borrow a book that is already checked out, the procedure can issue an error message like **"Book is not available"**. This prevents invalid actions and ensures that only correct data is inserted into the system.

Graph Database and Key Concepts

A graph database is a type of database that utilizes graph structure to represent and store data. In contrast to traditional relational databases, which organize information in tables and rows, a graph database captures data as nodes (or vertices) and the connection between them as edges. This innovative structure simplifies the modeling of complex many-to-many relationships and facilitates efficient navigation through these connections.

A vertex, often referred to as a node, serves as a representation of an entity or object within a graph. Each vertex has the capability to store information about the associated entity, including various attributes and properties. In my system, vertices symbolize different types of entities, such as: Users, these vertices represent individuals utilizing the system, with attributes like name, last name, email and user ID.

An edge signifies the connection or relationship between two vertices. Each edge has a specific direction, indicating the flow from one vertex to the other, and may also possess attributes that offer additional context regarding the relationship. In my system, edges are utilized to illustrate various types of relationships, including: when a user borrows a book, a connection is established between the user vertex and the book vertex. This connection includes attributes such as the borrow date, due date and return date.

A path consists of a sequence of vertices linked by edges, serving as a way to explore the relationships between entities. Depending on the direction of the connecting edges, paths can be categorized as either inbound or outbound.

Example of an inbound oath in my system:
Path: User → Borrow (inbound) → Book
In this scenario, the relationship is formed from the user to the book through the borrow action. since the edge directs towards the book, it is classified as an inbound path in relation to the book.

Example of an inbound path in my system:
Path: User → Borrow (outbound) → Book → Written By (outbound) → Author
In this scenario, the conection begins with User1 and extends to the book through the borrow edge. From the book, the connection then flows to the author via the written by edge. This path illustrates the relationship between a user, the book they have borrowed and the author of those books.

Attributes are properties associated with vertices and edges, providing additional context for each entity and relationship. They can take the form of simple key value pairs, such as user email, a loan date or the title of a book.

Graph Database Queries

The first query, **Show the books that User1 has borrowed and on which date**, retrieves a list of books that **User1** has borrowed, along with the **date each book was borrowed**. The query starts from the **Users** collection and follows an **OUTBOUND path** through the **Borrowed_Books** edges to the **Books** collection. For each connected book, the query returns

information about the **book** as well as the **borrow_date**, which is stored as an attribute on the edge. This query is useful for generating a user's **borrow history**, allowing the system to display which books a user has borrowed and on what date. It could be displayed on a user's profile page or used in an admin dashboard to track user activity.

The second query, **Find all the users who borrowed Book3**, identifies all users who have borrowed **Book3** and the **date they borrowed it**. The query starts from the **Books** collection and follows an **INBOUND path** through the **Borrowed_Books** edges to the **Users** collection. It returns the information about each **user** who borrowed the book, as well as the **borrow_date**. This query is useful for tracking book activity and understanding which users have interacted with a particular book. It can be used for **admin reports**, to notify users if a book is overdue, or to provide insights on the popularity of specific books.

Full Text Search in Relational/Document Databases

Full-Text Search is a **search technique** used in relational and document databases to find specific words or phrases within large text fields. Unlike basic searches that only match exact terms, Full-Text Search can match **partial words, phrases, and similar terms**, making it more effective for searching large text fields like **book titles, descriptions, and author bios**. It supports **natural language queries**, **fuzzy matching**, and can **rank results by relevance**, which provides a more user-friendly search experience.

In my system, Full-Text Search allows users to **search for books using keywords**. For example, if a user searches for "**data science**", the system can search through **book titles, descriptions, and categories** to find relevant results. Unlike a basic **SQL LIKE** query, Full-Text Search can match phrases like "**A Beginner's Guide to Data Science**", even if the user only types "**data science**". This provides more accurate and relevant search results.

To implement Full-Text Search, I would use **MATCH() AGAINST()** in relational databases like MySQL or **ArangoSearch** in ArangoDB. In ArangoDB, I would create a **View** that indexes key fields, such as **title, description, and category**, from the **Books collection**. When a user performs a search, the system scans these indexed fields and returns results ranked by relevance.

This approach makes search **faster, more accurate, and user-friendly**. By enabling natural language search and supporting partial matches, users can find books even if they don't

know the exact title. This improves **user experience**, ensures faster lookups, and makes it easier for users to discover relevant books in the system.

Efficiently Managing Large Documents

When working with document databases, it's common to encounter situations where a single document contains too much data. This often happens when a document includes large **nested arrays** or **embedded objects** that grow over time. For example, a **user document** may store an embedded list of all the books a user has borrowed. If the list grows too large, it can make updates, searches, and data management more difficult.

One major issue with this structure is **data redundancy**. If details about a book (like its title) are updated, every user document referencing that book must also be updated. This process is inefficient and can cause inconsistencies. To solve this, the large document is **split into smaller, more focused documents**, similar to normalization in relational databases. Instead of embedding all book details in the user document, I would create a **Books collection** and store only the **book IDs** in the user's document. This approach reduces redundancy and allows updates to book details to happen in a single location.

To manage the connections between smaller documents, I use **reference-based connections** or **graph-based connections**. In a reference-based approach, a document stores the **_id** of related documents, allowing for fast lookups and more efficient queries. For more complex relationships, a **graph-based approach** is more effective. Documents become **nodes** and their relationships are represented by **edges**. This method is useful for **many-to-many relationships** and **multi-level queries**, like tracing which users have borrowed books from specific authors.

By breaking down large documents into smaller, related documents, I ensure **better efficiency, reduced redundancy, and easier maintenance**. This approach also enables **faster lookups and updates**, since changes to one document automatically reflect in related documents.

What is an In-Memory Database?

An in-memory database (IMDB) is a unique type of database that stores all its data in RAM (Random Access Memory) instead of on a traditional disk. This design enables drastically quicker read and write operations compared to conventional disk-based databases. Unlike relational databases, which focus on achieving high speed and minimal latency. An in-memory database like Redis can significantly enhance the speed, performance and overall user experience of my system. By managing user sessions, caching frequently accessed data and enforcing request limits, Redis can help reduce the burden on the primary relational database and speed up key features in the system. Here are some specific areas within my system where Redis would be effectively implemented.

User Session Management:

To efficiently store and manage user login sessions and role data, we use Redis, which ensures quick logins and secure authentication. Upon logging in, a user's session information—including their user ID, email, and role—is saved in Redis, with each session assigned a unique key. Additionally, each session is given a time-to-live (TTL) so that if a user remains inactive for a specified duration, such as one hour, the session will automatically expire, resulting in the user being logged out.

Caching Frequently Accessed Data:

To reduce the load on the main relational database and speed up data retrieval. When users search for books or view a list of authors, the system can cache this information in Redis. Instead of querying the main database every time, the system would check Redis first. If the search data is found in Redis, it is displayed instantly. If it is not found, the system queries the main database, and the result is then stored in Redis for future use.

Rate Limiting and Request Throttling:

To prevent users from overloading the system with too many requests, particularly when searching for books or authors. I would monitor how many times each user sends a search request in a short time period. If a user sends too many requests, such as 100 requests per minute, the system would block them temporarily.

Denormalizing a Relational Database

Denormalization can be beneficial in situations where query performance needs to be improved, especially in **read-heavy systems** where the database is frequently queried to display information to users.

One situation where denormalization would be beneficial in my system is in the **Borrowed_Books** table. Normally, if I wanted to display a list of borrowed books along with the user's name and the book's title, I would need to join the **Borrowed_Books**, **Users**, and **Books** tables. This query can become slow as the number of records in the system grows. To address this, I could denormalize the **Borrowed_Books** table by storing the **user_name** and **book_title** directly in it. By doing so, I avoid the need for multiple JOINs, and I can retrieve all the necessary information with a single query from the **Borrowed_Books** table. This approach makes it easier to generate reports and improves response time for end users.

Another situation where denormalization would be useful is for tracking **borrow counts**. In a normalized database, calculating how many times a specific book has been borrowed would require counting the number of entries for that book in the **Borrowed_Books** table. This query becomes slower as the number of borrow records grows. To solve this, I could add a **borrow_count** column directly to the **Books** table. Each time a book is borrowed, the system would increment the **borrow_count**. This allows me to see how often a book has been borrowed without needing to query the entire **Borrowed_Books** table. It also simplifies the process of generating dashboards or analytics, as the total count is already stored in the **Books** table.

Vertical vs Horizontal Scaling

As a system grows, it becomes necessary to **scale the database** to handle larger datasets and increased user requests. There are two main methods to achieve this: **Vertical Scaling (Scaling Up)** and **Horizontal Scaling (Scaling Out)**. Each method has unique advantages and is suited for specific situations.

Vertical Scaling involves increasing the capacity of a single server by upgrading its **CPU, RAM, storage, or network capacity**. This approach focuses on making a single server more

powerful rather than distributing the workload. For example, if I notice that querying the **Borrowed_Books** table takes too long due to increased data, I could upgrade the server's **RAM** to enable faster in-memory operations. Vertical scaling is simpler to implement since no data redistribution is required. However, it has limitations — there is a physical limit to how much hardware can be upgraded, and since all operations depend on a single server, there is a **single point of failure**. If the server crashes, the entire system could go offline.

Horizontal Scaling focuses on distributing the workload across **multiple servers**. Instead of upgrading a single server, additional servers (nodes) are added to **split the data and requests**. This approach is more effective for large datasets and high-traffic systems. For example, if the number of users and Borrowed_Books records grows significantly, I could distribute the **Users**, **Books**, and **Borrowed_Books** collections across multiple ArangoDB nodes. Each server would handle a portion of the records, reducing the workload on individual servers. Horizontal scaling allows for **unlimited growth**, as more servers can be added when needed. It also provides **fault tolerance**, since if one server fails, the other servers can continue running, ensuring system availability.

Both vertical and horizontal scaling are essential at different stages of system development. **Vertical scaling** is ideal during the early stages, when the system handles fewer users and smaller datasets. It is simple to manage and requires fewer resources. However, as the system grows, **horizontal scaling** becomes more suitable, especially when user traffic increases and the number of **Borrowed_Books** records grows into the millions. By using multiple ArangoDB nodes, I can distribute data across multiple servers, ensuring no single server is overwhelmed.

A balanced strategy would be to start with **vertical scaling** during initial development, as it is cost-effective and simple to implement. As the system grows, I would transition to **horizontal scaling** to handle larger datasets, increased user requests, and ensure fault tolerance. This approach allows for scalability, better performance, and high system availability as the system continues to expand.