Dental Clinic DBMS.

June 2022

Kareen Ziadat

Contents

Problem definition, user and system requirement, ERD

Problem definition

In the last year, a dental clinic received many complaints including double booking, booking the wrong doctor and accidentally cancelled appointments. The dental clinic has multiple dentists and find it hard to manage their appointments. They want to design a database that allows the secretary to keep track of the dental appointments.

User requirements

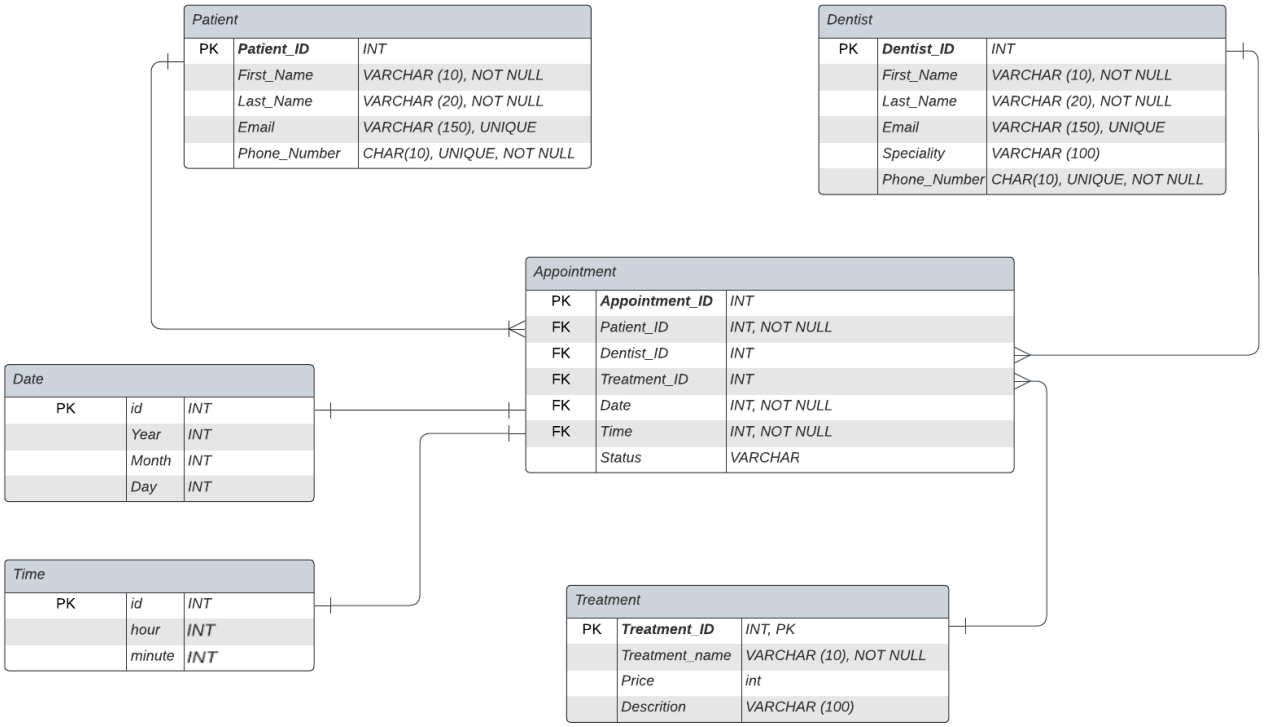
The secretary should be able to:

- View today's appointment schedule.
- View all the appointments.
- View the details of the dentists available.
- View all the treatments offered by the dental clinic.
- Create a new dental appointment.
- Change the details of an already created appointment.
- Cancel an appointment.

System requirements

- Ensure the primary key is unique, not null.
- Ensure the foreign key is an existing primary key in the linked table.
- Ensure unique values are unique when inserting new records.
- Ensure any other validations like the length.
- Data shouldn't be lost when the program stops running.
- Memory allocation should be efficient and dynamic.
- Should handle processing large amounts of data by applying threading techniques.
- The data structure should be optimized for fast data retrieval through indexing.
- The interface (menu) should not be constricting to the user and should cover all functionalities.
- Guarantee of durability ensuring the results of completed transaction will never be lost.

ERD


Patient
PK Patient_ID INT
First_Name VARCHAR (10), NOT NULL
Last_Name VARCHAR (20), NOT NULL
Email VARCHAR (150), UNIQUE
Phone_Number CHAR(10), UNIQUE, NOT NULL
Dentist
PK Dentist_ID INT
First_Name VARCHAR (10), NOT NULL
Last_Name VARCHAR (20), NOT NULL
Email VARCHAR (150), UNIQUE
Speciality VARCHAR (100)
Phone_Number CHAR(10), UNIQUE, NOT NULL
Appointment
PK Appointment_ID INT
FK Patient_ID INT, NOT NULL
FK Dentist_ID INT
FK Treatment_ID INT
FK Date INT, NOT NULL
FK Time INT, NOT NULL
Status VARCHAR
Date
PK id INT
Year INT
Month INT
Day INT
Time
PK id INT
hour INT
minute INT
Treatment
PK Treatment_ID INT, PK
Treatment_name VARCHAR (10), NOT NULL
Price int
Descrition VARCHAR (100)

Architecture components of DBMS

The architecture described consists of components (Query compiler, transaction manager, DDL compiler, Execution engine, logging and recovery, concurrency control, index/file/record manager, buffer manager, buffers, lock table, Storage manager and the storage) through which data flows.

Commands for the DBMS are either given by a user or the database administrator.

1. Users can enter commands directly or through an application program to request or modify data in a database. These commands are received by the **query compiler**, which is responsible for processing and organizing them. The handling of data manipulation language (DML) systems involves two main processes: transaction processing, which is managed by the **transaction manager** component, and query answering, which is performed by the **query compiler** and **execution engine** components.
2. The database administrator makes changes to the structure of the database using the DDL commands. These commands are processed and organized by the **DDL compiler**. The processed commands are then executed by the **execution engine**, and the necessary updates are made to the database's structure. The **index/file/record manager** modifies the database's metadata.

The components

Transaction manager

Multiple operations can be carried out in the same transaction, these operations should be able to execute directly, and in complete isolation of each other without the results of any transactions being lost.

The application or user sends the commands to the transaction manager telling it when the transaction should be started and completed. When the transaction manager receives the commands it should perform logging, concurrency control and deadlock resolution to meet the system requirements of the DBMS.

The transaction manager will send the data to the logging and recovery component to ensure the durability of the DBMS. Moreover, the transaction manager will send the data to the concurrency control component to allow execution of transactions in insulation. lastly, it also handles topics such as deadlock identification and resolution. **Deadlocks** occur when two or more transactions wait forever for resources held by each other, causing the system to come to a halt. The transaction manager is responsible for resolving this issue by cancelling one or more transactions.

I employed threading technique to process big amounts of records when viewing the entire table. I let a separate thread process each node in the linked list (one thread will be used to read each record in the table

Logging and recovery

The logging and recovery component follows a policy that protects data integrity and recoverability in the case of a system crash or failure.

When transactions alter data, the logging component records the changes in a log file. The log file maintains a record of all database modifications (old and new data). This logging is usually performed in the background, concurrently with transaction execution.

When transactions modify data, the changes are first written to the memory's database buffers. The logging component talks with the buffer manager to ensure that the modified data in the buffers is sent to disk ensuring that the changes are reflected.

In the event of a system crash or failure, the log file is used by the recovery process to restore the database to a consistent state.

Concurrency control

The concurrency control component, commonly known as the scheduler, assures the proper and coordinated execution of numerous concurrent transactions. It keeps consistency and isolation to produce the same results as if they were executed sequentially.

A locking mechanism is implemented via the concurrency control component. Transactions that cause conflicting modifications on the same data are prevented from being executed by the locks. (One programming techniques used to achieve concurrency control)

In addition to locks, concurrency control employs scheduling algorithms to determine the order in which transactions are executed. To avoid data inconsistencies, this guarantees that conflicting operations are serialized appropriately.

Query compiler (DML)

The query compiler is in charge of **parsing and optimizing** database manipulation language (DML) user requests. It takes the user's query and converts it into an optimal plan for retrieving or altering data by performing numerous operations.

The **query parse**r validates the query by analyzing its syntax and structure. then generates a parse tree that represents the query's syntactic structure.

The **query preprocessor** performs semantic checks on the query after it has been processed, verifying that the referred database objects (tables, columns, etc.) exist and that the query corresponds to the provided constraints and data types. It also checks for access control. Following the semantic checks, the query preprocessor converts the parsed tree to a tree of algebraic operations.

The query optimizer performs query optimization. The optimizer uses the algebraic expression tree to identify the most efficient sequence of operations for retrieving or altering the data. It uses indexes and cost models to estimate the execution time of various query plans. The optimizer seeks the best query plan that reduces total execution costs while maximizing performance.

The execution engine

The execution engine is in charge of carrying out the steps indicated in the query plan generated by the query compiler. It communicates with multiple DBMS components to efficiently access and alter data. It communicates with the buffer manager, which oversees caching data pages in memory, decreasing the need for frequent disk operations (faster data retrieval).It also interacts with the Concurrency control component to ensure data consistency and avoid conflicts between concurrent transactions. Furthermore, the execution engine communicates with the log manager to ensure that database updates are properly logged.

It may also communicate with the index manager in order to access and modify index structures.

Buffer manager

The buffer manager is in charge of managing the main-memory buffers that are used to cache data from disk.

These buffers store data pages that have been read from disk. It keeps track of the available buffers as well as their status (occupied or free). When a data request is made and the desired data is not found in the buffers, the buffer manager contacts the storage manager.

Furthermore, the buffer manager flushes updated pages from the buffer pool to disk on a regular basis. Changes made in the buffers are saved to disk, ensuring data consistency and durability.

Programming technique used: Dynamic memory allocation was employed by dynamically allocating memory when inserting a record using malloc. I didn't use realloc as I am using a linked list data structure which dynamically expands and decreases by freeing memory.

Storage manager

The storage manager oversees data storage management. It manages disk space, file organization (controls where data is placed on the disk), data structure management, and data integrity.

When a data request is performed, it guarantees that the necessary data portions are sent from the disk into the main-memory buffers (page). This data movement between secondary storage (disk) and main-memory buffers is critical for fast data access and retrieval.

Programming technique used:

To ensure efficient and reliable data storage, the storage manager component uses programming techniques such as file system operations (e.g., file creation, deletion), disk space management (e.g., allocation, deallocation), data structure management (e.g., organizing data on disk, indexing structures), and data integrity checks (e.g., checksums, validation).

To get the relational structure of a database, I relied on creating a struct for each table. I used a linked list data structure to store the records (instants of the structs). I created a file management system that allowed for record storage and retrieval even after the program had stopped running by creating a file for each struct and writing the records (data in the linked list) onto the file after each insert function calling. When the program stops execution, the code will read from these files and save them in the linked lists effectively maintaining the records of the database.

I dynamically allocated and deallocated disk space which allowed for efficient disk space management. When the size of the data decreases, the storage manager reclaims the unused disk space and makes it available for future use using the free function.

I relied on the use of pointers as I am using a linked list data structure so to link the nodes to each other, pointers were used.

Buffers

Buffers are main-memory areas that are used to temporarily store data pages read from disk. These buffers serve as a cache between the disk and the CPU, providing faster access. Data is often read from or written back to the disk in blocks or pages rather than individual records.

Index/file/record manager

The Index/File/Record Manager component is responsible for managing the organization, storage, and retrieval of data within the database.

The Index Manager manages the creation, maintenance, and utilization of indexes to optimize query performance. Indexes improve the efficiency of data retrieval operations by providing quick access to data based on specific search criteria. Indexing can be done on a single-level (primary, clustering, or secondary indexing) or multilevel indexing (B Trees, B+ Trees). Hashing is widely used to create hash indexes. A hash function is used in hash indexes to map keys to specified positions in the index structure.

The File Manager handles tasks such as file creation, deletion, and organization. It manages the data block layout, and allocation of disk space for files.

The Record Manager is responsible for managing individual records within a file, so it must interact with the File Manager to read and write data at the record level.

Programming technique used:

I was able to use a binary search technique for efficient data retrieval since the primary key of my tables were sorted ascendingly as they were autoincremented. This considerably increased the speed and efficiency of querying data, as well as all actions that rely on data retrieval.

The file creation programming techniques was also used and record-level operations were employed through reading, writing.

Overall

In my implementation of the storage manager component of the DBMS, I used structs to represent each relation in the database. Structs make it simple to specify and arrange the attributes and metadata associated with a relationship. I was able to represent individual records within the relations and manipulate them quickly by constructing instances of these structs.

I picked a linked list data structure to hold the data.

Because of the necessity for dynamic memory management, quick insertion and deletion operations, flexibility in record manipulation, and ease of traversal. These elements contribute to the overall efficacy and dependability of the DBMS storage management component.

To add, I made sure to validate that the user inputs adhere to the database constraints to ensure integrity.

SQL statements

SQL queries	CRUD statement
SELECT a.Appointment_ID, p.First_Name, p.Last_Name, d.First_Name, d.Last_Name, t.Treatment_Name, a.Date, a.Time, a.Status FROM appointments a JOIN patients p ON a.Patient_ID = p.Patient_ID JOIN dentists d ON a.Dentist_ID = d.Dentist_ID JOIN treatments t ON a.Treatment_ID = t.Treatment_ID WHERE a.Date = CURDATE();	viewSchedule
SELECT * FROM patients WHERE Patient_ID = 1; /*the patient id filled depends on the user input*/	findPatientByID

SELECT * FROM dentists WHERE Dentist_ID = 1; /*the dentist id filled depends on the user input*/	findDentistByID
SELECT * FROM treatments WHERE Treatment_ID = 1; /*the treatment id filled depends on the user input*/	findTreatmentByID
SELECT * FROM appointments WHERE Appointment_ID= 1; /*the appointment id filled depends on the user input*/	findAppointmentByID
INSERT INTO patients (First_Name, Last_Name, Email , Phone_Number) VALUES ('John', 'Doe', 'john@example.com', '1234567890'); /*the data filled depends on the user input*/	addPatient
INSERT INTO dates (day, month, year) VALUES (20,6,2023); /*the data filled depends on the user input*/	newDate
INSERT INTO times (hour, minute) VALUES (3,51); /*the data filled depends on the user input*/	newTime

INSERT INTO patients (First_Name, Last_Name, Email ,Phone_Number) VALUES ('John', 'Doe', ' ' ,'1234567890'); /*the data filled depends on the user input*/ INSERT INTO dates (day, month, year) VALUES (20,6,2023); INSERT INTO times (hour, minute) VALUES (3,51); INSERT INTO appointments (Patient_ID, Dentist_ID, Treatment_ID, Date, Time, Status) VALUES (12, 2, 3, 1, 1, 'booked'); /*patient ID is from the one previously created, data filled depends on user input*/	createAppointmentNewPatient Case 1
INSERT INTO patients (First_Name, Last_Name, Email ,Phone_Number) VALUES ('John', 'Doe', ' ' ,'1234567890'); /*the data filled depends on the user input*/ INSERT INTO dates (day, month, year) VALUES (20,6,2023); INSERT INTO times (hour, minute) VALUES (3,51); INSERT INTO appointments (Patient_ID, Treatment_ID, Date, Time) VALUES(1, 2, 1, 1); /*patient ID is from the one previously created, data filled depends on user input*/	createAppointmentNewPatient Case 2

INSERT INTO patients (First_Name, Last_Name, Emai ,Phone_Number) VALUES ('John', 'Doe', ' ' ,'1234567890'); /*the data filled depends on the user input*/ INSERT INTO dates (day, month, year) VALUES (20,6,2023); INSERT INTO times (hour, minute) VALUES (3,51); INSERT INTO appointments (Patient_ID, Dentist_ID, Date, Time) VALUES(1, 2, 1, 1); /*patient ID is from the one previously created, data filled depends on user input*/	createAppointmentNewPatient Case 3
INSERT INTO patients (First_Name, Last_Name, Emai ,Phone_Number) VALUES ('John', 'Doe', ' ' ,'1234567890'); /*the data filled depends on the user input*/ INSERT INTO dates (day, month, year) VALUES (20,6,2023); INSERT INTO times (hour, minute) VALUES (3,51); INSERT INTO appointments (Patient_ID, Date, Time) VALUES(1, 1, 1); /*patient ID is from the one previously created, data filled depends on user input*/	createAppointmentNewPatient Case 4

INSERT INTO dates (day, month, year) VALUES (20,6,2023); INSERT INTO times (hour, minute) VALUES (3,51); INSERT INTO appointments (Patient_ID, Dentist_ID, Treatment_ID, Date, Time) VALUES (12, 2, 3, 1, 1); /* data filled depends on user input*/	createAppointmentPatientExists Case 1
INSERT INTO dates (day, month, year) VALUES (20,6,2023); INSERT INTO times (hour, minute) VALUES (3,51); INSERT INTO appointments (Patient_ID, Treatment_ID, Date, Time) VALUES(1, 2, 1, 1); /*data filled depends on user input*/	createAppointmentPatientExists Case 2
INSERT INTO dates (day, month, year) VALUES (20,6,2023); INSERT INTO times (hour, minute) VALUES (3,51); INSERT INTO appointments (Patient_ID, Dentist_ID, Date, Time, Status) VALUES(1, 2, 1, 1); /* data filled depends on user input*/	createAppointmentPatientExists Case 3

INSERT INTO dates (day, month, year) VALUES (20,6,2023); INSERT INTO times (hour, minute) VALUES (3,51); INSERT INTO appointments (Patient_ID, Date, Time) VALUES(1, 1, 1); /* data filled depends on user input*/	createAppointmentPatientExists Case 4
INSERT INTO dates (day, month, year) VALUES (22,6,2023); UPDATE appointments SET Date = 2 WHERE Appointment_ID = 1; /*modified date and appointment id depends on the user*/	changeAppointmentDate case 1 of changeAppointmentDetails
INSERT INTO times (hour, minute) VALUES (4,0); UPDATE appointments SET Time = 2 WHERE Appointment_ID = 1; /*modified time and appointment id depends on the user*/	changeAppointmentTime case 1 of changeAppointmentDetails
UPDATE appointments	changeAppointmentDoctor

SET Dentist_ID = 3 WHERE Appointment_ID = 1; /*modified dentist id and appointment id depends on the user*/	case 1 of changeAppointmentDetails
UPDATE appointments SET Treatment_ID = 3 WHERE Appointment_ID = 1; /*modified treatment id and appointment id depends on the user*/	changeAppointmentTreatment case 1 of changeAppointmentDetails
INSERT INTO dates (day, month, year) VALUES (21,6,2023); UPDATE appointments SET Date = 2 WHERE Patient_ID= 1; /*modified date and patient id of appointment depends on the user*/	changeAppointmentDate case 2 of changeAppointmentDetails
INSERT INTO times (hour, minute) VALUES (4,00); UPDATE appointments SET Time = 2 WHERE Patient_ID= 1; /*modified time and patient id of appointment depends on the user*/	changeAppointmentTime case 2 of changeAppointmentDetails
UPDATE appointments SET Dentist_ID = 3	changeAppointmentDoctor

WHERE Patient_ID= 1; /*modified dentist id and patient id of appointment depends on the user*/	case 2 of changeAppointmentDetails
UPDATE appointments SET Treatment_ID = 3 WHERE Appointment_ID = 1; /*modified treatment id and patient id of appointment depends on the user*/	changeAppointmentTreatment case 2 of changeAppointmentDetails
DELETE FROM appointments WHERE Appointment_ID = 1; /*appointment id depends on the input of the user*/	deleteAppointmentByID case 1 of cancelAppointment
DELETE FROM appointments WHERE Patient _ID = 1; /*patient id depends on the input of the user*/	deleteAppointmentByPatientID case 2 of cancelAppointment
SELECT * FROM appointments;	printAppointments
SELECT * FROM dentists;	printDentists
SELECT * FROM treatments;	printTreatments

Concurrency control techniques

Concurrency control is a critical component of Database Management Systems (DBMS) that maintains data integrity and correctness in multi-user scenarios. It is concerned with managing concurrent access to shared data by numerous transactions to avoid inconsistencies and data integrity breaches. I will describe three types of concurrency control mechanisms: locking-based, timestamp-based, and optimistic concurrency control.

1. Locking-Based: These protocols require transactions to get appropriate locks before reading or writing data to ensure isolation and avoid concurrent issues. It does this by isolating a particular transaction to a single user. Shared locks (S-locks) and exclusive locks (X-locks) are the two most frequent forms of locks. Shared locks (S-locks) allow data to be read by many transactions simultaneously. Since they simply need read access and do not alter the data, transactions holding shared locks do not interfere with one another. Exclusive locks (X-locks) on the other hand stop other transactions from accessing the locked data and modifying it. An exclusive lock can only be held by one transaction at a time. When a transaction wants to modify the data, exclusive locks are utilized to prevent concurrent access by other transactions.
   Advantages:
   1. It is simple to understand and apply.
   2. It is compatible with conventional databases and applications.
   3. Locks can be imposed at many levels from entire tables to individual data items.

   Disadvantages:
      1. If locks are not managed appropriately, deadlocks happen resulting in many transactions being blocked indefinitely.
      2. The use of locks imposes additional overhead due to lock acquisition, release, and conflict detection.
      3. The degree of parallelism is less because transactions frequently have to wait for locks.

   Lock-based concurrency management is employed when conflicts between transactions are not frequent or intense.

2. Timestamp-Based: it assigns timestamps to transactions to set an order of execution and manages conflicts.
   Advantages:

1. Timestamps ensure that transactions are completed sequentially, ensuring that the final outcomes are consistent.
2. Non-conflicting timestamp transactions can run concurrently, enhancing throughput.
3. Deadlock issues can be prevented by using a rigorous timestamp order.

Disadvantages:

1. Timestamp management can be complicated by assigning unique and monotonically rising timestamps.
2. If two transactions have the same timestamp, further conflict resolution procedures are required.
3. Certain sorts of conflicts may be difficult to resolve using timestamp-based strategies.

Often employed in systems where strict serializability is required, such as banking applications and reservation systems

3. Validation-Based: Also known Optimistic Concurrency Control, considers that conflicts are uncommon and detects them during transaction validation.

Advantages:

1. Transactions can run concurrently without gaining locks, enhancing overall system throughput.
2. Since locks are not held throughout execution, reducing contention (numerous processes or instances compete for access to the same index or data block.) among transactions.
3. Because lock acquisition and release are not required, the associated overhead is reduced.

Disadvantages:

1. Conflicting transactions must be aborted and retried, incurring additional expense.
2. Implementing conflict identification and resolution processes can be difficult and time-consuming.
3. In circumstances with recurrent disagreements, optimistic tactics may be less effective.

Employed in cases where conflicts are expected to be few or where the cost of aborting and retrying a transaction is lower than the cost of managing locks. It's used in e-commerce platforms and content management systems.

Data Transfer Evaluation and System Workflow Analysis

Data flow between the databases components:

Outline of which components I used and how.

Transaction Manager: Information is transferred from the linked list of records to the transaction manager's threads. Each thread is in charge of handling a particular node or record in the linked list. The linked list is retrieved by the transaction manager in order to obtain the record related to each node. On the record within the thread, the relevant actions, such as data reading, are carried out.

Buffer manager: Data transfer in the buffer manager essentially entails dynamic memory allocation for managing the linked list structure and adding data. The buffer manager uses the malloc function to dynamically allocate memory when a record is inserted, and it also generates a new node in the linked list to represent the record.

Storage Manager: In the storage manager, information is transferred between the file system and the linked list data structure. The storage manager puts the record's data onto a file specific to the related struct or table whenever a record is added or updated. Data persistence even after program execution has ended is ensured by this transfer. The storage manager reads the data from the files and saves it back into the linked lists when the program begins execution, thereby rebuilding the database records in memory.

Data transfer in this component's index/file/record manager involves effective record retrieval and manipulation based on primary keys and file management operations. The primary key is utilized to locate records using the binary search approach. To access the data in the files, record-level operations like reading and writing are used. For these activities, data is moved back and forth between the files and the manager.

The storing, retrieval, and manipulation of records are made easier by a variety of data structures (linked lists, files), processes (threads), and other elements of the DBMS architecture. The coordination and effective management of the database's information are made possible by the mobility of data.

Data movement between database components

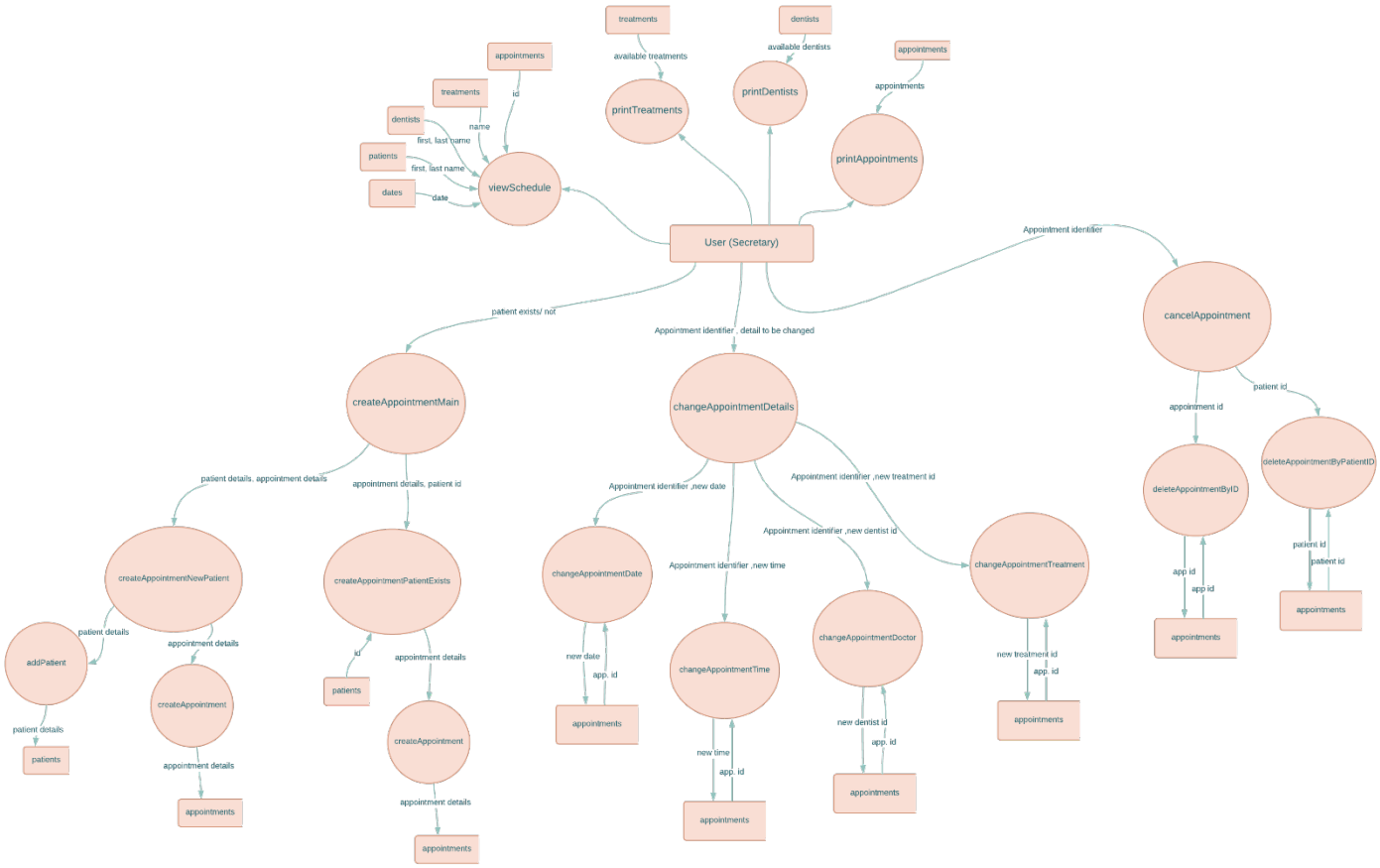
Transfer of data from linked list to transaction manager: Transfers data from the linked list of records to transaction management threads. Each thread in the linked list is in charge of managing a particular node or record. The transaction manager gets the linked list to obtain the record associated with each node, then executes the necessary operations on the record within the thread, such as data reading.

Transfer of data from buffer manager to storage manager: Handles dynamic memory allocation (malloc) for managing the linked list structure and adding data onto the storage through the storage manager.

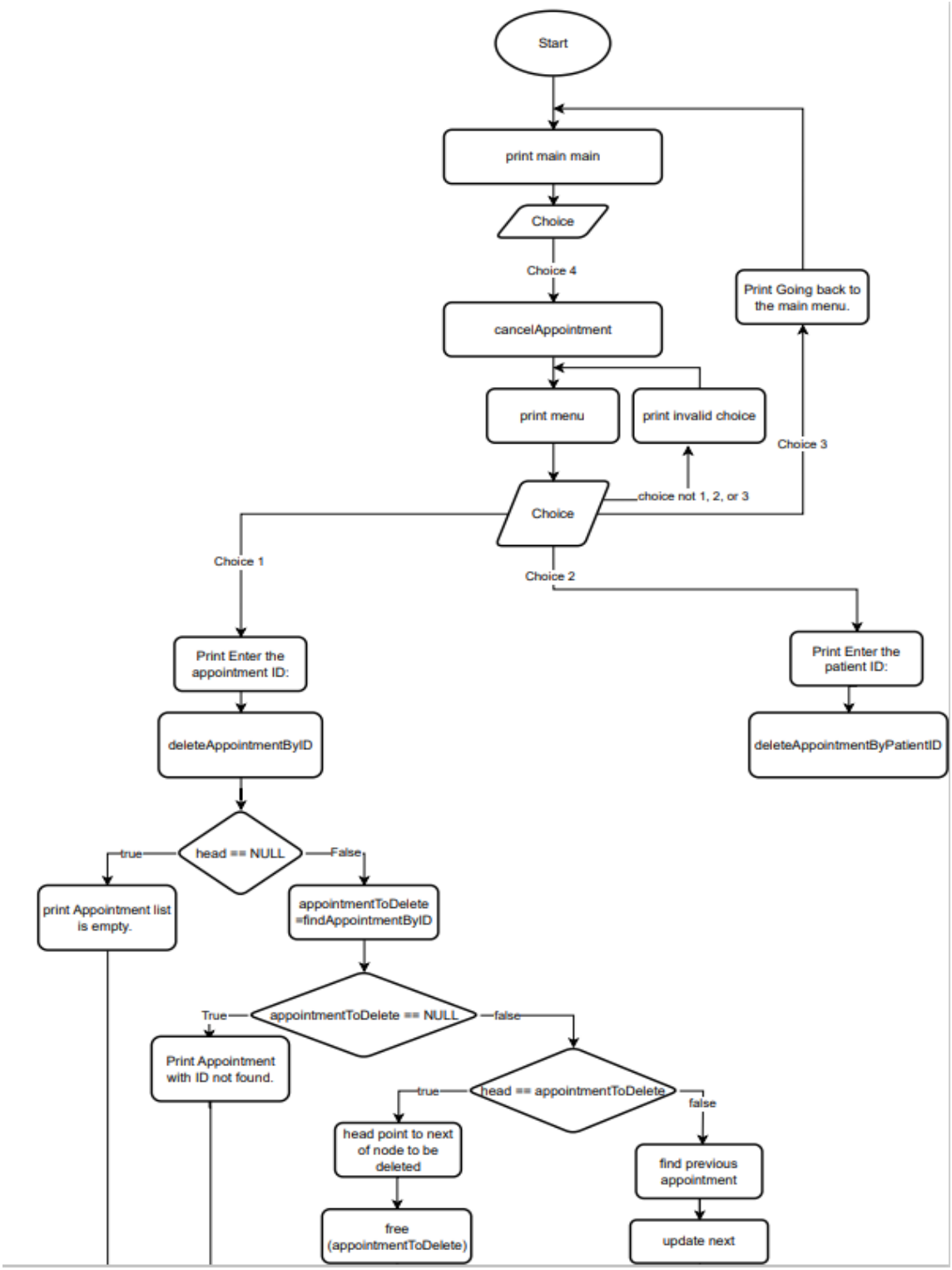
File manager and storage manager: Since the File Manager is in charge of maintaining the files used to store database records, it interacts with the Storage Manager. The File Manager works with the Storage Manager to store the data in particular files connected to the corresponding struct or table when records are added to or changed.

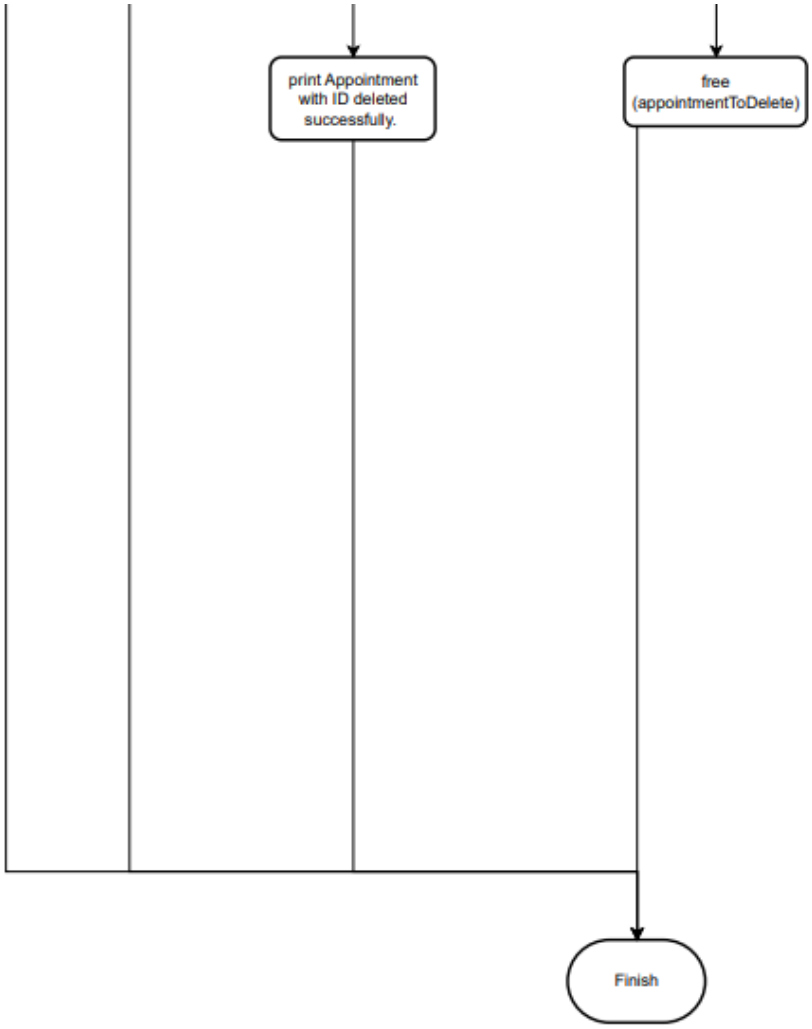
Storage manager to and from disk storage: Transfers information between the file system and the linked list data structure. In order to ensure data durability even after program execution has ceased, it accomplishes this by writing the structs' data to the corresponding file when a record is inserted. When the software starts running, it reads the data from the files and writes it back into the linked lists, recreating the database records in memory.

Date flow diagram:



Flowchart for cancelling an appointment.




print Appointment with ID deleted successfully.
free (appointmentToDelete)
Finish

Testing plan

Data Validation

Test Description: Ensure that it only allows addition of patient when the email is unique.
Test: Add a patient with a unique email.
Expected: The system successfully adds the patient to the database.

Test Description: Ensure that it only allows addition of patient when the phone number is unique.
Test: Add a patient with a unique phone number.
Expected: The system successfully adds the patient to the database.

Test Description: Ensure that it only allows addition of patient when the email is unique.
Test: Add a patient with a duplicate email.
Expected: The system displays an error message indicating that the email is already in use and prevents the addition of the patient.

Test Description: Ensure the appointment is created at the time and date specified and that it correctly searches for the patient id, treatment id, and the dentist Id and shows an error if they don't exist.
Test: Create an appointment with specific time, date, patient id, treatment id, and dentist id.
Expected: The system successfully creates the appointment with the provided details. If any of the ids don't exist, the system displays an error message.

Test Description: When creating an appointment for a new patient, make sure the correct id of the new patient is assigned to the appointment.
Test: Add a new patient and create an appointment for that patient.
Expected: The system successfully adds the new patient to the database and assigns the correct patient id to the appointment.

Data Modification

Test Description: Make sure that it changes the date of the appointment correctly by either searching by the appointment id or by the patient id. It should also show an error when the appointment id or the patient id doesn't exist.
Test: Change the date of an existing appointment using either the appointment id or patient id.
Expected: The system successfully updates the appointment's date. If the provided id doesn't exist, the system displays an error message.

Test Description: Make sure that it changes the time of the appointment correctly by either searching by the appointment id or by the patient id. It should also show an error when the appointment id or the patient id doesn't exist.
Test: Change the time of an existing appointment using either the appointment id or patient id.
Expected: The system successfully updates the appointment's time. If the provided id doesn't exist, the system displays an error message.

Test Description: Make sure that it changes the treatment id of the appointment correctly by either searching by the appointment id or by the patient id. It should also show an error when the appointment id or the patient id doesn't exist.
Test: Change the treatment id of an existing appointment using either the appointment id or patient id.
Expected: The system successfully updates the appointment's treatment id. If the provided id doesn't exist, the system displays an error message.

Test Description: Make sure that it changes the dentist id of the appointment correctly by either searching by the appointment id or by the patient id. It should also show an error when the appointment id or the patient id doesn't exist.
Test: Change the dentist id of an existing appointment using either the appointment id or patient id.
Expected: The system successfully updates the appointment's dentist id. If the provided id doesn't exist, the system displays an error message.

Test Description: Ensure the appointment with the appointment id is cancelled correctly.
Test: Cancel an existing appointment using the appointment id.
Expected: The system marks the appointment as cancelled and updates the status accordingly.

Test Description: Ensure the appointment with the patient id is cancelled correctly.
Test: Cancel an existing appointment using the patient id.
Expected: The system marks the appointment as cancelled and updates the status accordingly

Test Description: Delete the same appointment (by appointment id) and check if it is still found in today's schedule.
Test: Delete an appointment using the appointment id and verify if it is still listed in today's schedule.

Expected: The system should successfully delete the appointment and remove it from today's schedule.

Data Retrieval

Test Description: Make sure that it is only printing the appointments of today when choosing option 1 of the menu.
Test: Select option 1 to view appointments and verify that only today's appointments are displayed.
Expected: The system correctly displays only the appointments scheduled for the current date.

.
Test Description: Indexing should work properly and retrieve the correct record.
Test: Search for appointments using different indexing methods (e.g., by appointment id, patient id) and verify that the correct records are retrieved.
Expected: The system correctly retrieves the appointment records based on the provided indexing method.

File I/O

Test Description: Make sure file reading and writing is working.
Test: Save some appointments to a file and then read the file to retrieve the appointments.
Expected: The system successfully saves the appointments to the file and can read the file to retrieve the saved appointments.

Performance

Test Description: Test if viewing functions take a long time.
Test: Measure the time it takes to view the appointments and assess the performance.
Expected: The system should display the appointments within an acceptable timeframe, indicating good performance.

Data Persistence

Test Description: Try adding an appointment with a new patient and set the date to today's date. Stop the program, rerun, press option 1 to see if the patient appointment was added.
Test: Add an appointment for a new patient with today's date, stop the program, restart it, and verify if the appointment is still present when selecting option 1.
Expected: The system should persist the added appointment, and it should still be present when the program is restarted, and the appointments are viewed.

Evaluation of the effectiveness of the design

The ERD provided is well-designed and effectively handles the problem definition, user requirements, and system requirements.

First: Problem Definition.

By designing the database in a way that allows the secretary to track and manage appointments effectively, the ERD provides a solution to the issue definition of managing dental appointments. The ERD particularly addresses the identified concerns as follows:

The ERD addresses the issue of double booking by utilizing the Appointments table, which includes foreign keys for the date, time, and dentist. These keys establish a one-to-one relationship with the Dates and Times, and dentists' tables, where the date ID, time ID, and Dentist ID are primary keys, respectively. By enforcing uniqueness constraints on the date and time records for each dentist ID, the ERD ensures that only one instance of a specific date and time combination can exist for a specific doctor while also allowing creation of an appointment for a different dentist at the same date and time as another appointment. Consequently, this prevents the possibility of double booking by disallowing multiple appointments with the same date, time, and dentist in the database.

To effectively address the issue of booking the wrong doctor, the ERD establishes a one-to-one relationship between the Appointments and Dentists tables. This relationship is facilitated by the Dentist_ID foreign key in the Appointments table, which references the Dentists table. The Dentists table contains essential information about each dentist, such as their name, specialty, and unique Dentist_ID. As a result, the ERD ensures that each appointment is assigned to the relevant dentist. When scheduling a new appointment, the secretary can view the Dentists table to see a list of dentists' name, specialty, and id of that dentist to correctly choose the id for the patient's appointment.

Having a database in place that effectively manages the appointments provides a more robust solution that traditionally writing on pen and paper as it eliminates the risk of human error like misplaced or lost paper records (accidental cancelled appointments).

Furthermore, the database assures accurate record-keeping, optimizes scheduling operations, enhances data integrity, allows for insightful analysis, and provides data durability and security.

Second: User requirements.

The ERD efficiently solves user needs by providing the write set of attributes and functionalities that meet the secretary's needs in managing dental appointments. Here's a full breakdown of how the ERD addresses each user requirement:

1. View today's appointment schedule: The Appointments table includes attributes for each appointment's date and time. The secretary can access and view all appointments booked for the current day by querying the database with the current date.

2. View all appointments: All appointment records are stored in the Appointments table. The secretary can retrieve and examine a detailed list of all appointments by querying the table, which includes allocated patient details, dentists, treatments, and appointment status.

3. View details of dentists available: The Dentists table includes details about each dentist, such as their names, specialties, and contact information. The secretary can view the whole list of dentists and their details by accessing the Dentists table, allowing them to make educated judgments when assigning dentists to appointments.

4. View all treatments offered by the dental clinic: The Treatments table contains information on the dental clinic's treatments, such as treatment names, pricing, and descriptions. The secretary can access and read the full list of treatments provided at the clinic by querying the Treatments table.

5. Create a new dental appointment: The Appointments table is where the secretary can create new appointments. The secretary can enter the relevant information, such as the patient's ID, the dentist's ID, the treatment ID, and the appointment date and time, assuring correct record generation and assignment. It also prevents double bookings.

6. Change the details of an already created appointment: The Appointments table allows the secretary to search for and change existing appointment information. The secretary can accept rescheduling or other changes by amending the relevant attributes, such as the assigned dentist, treatment, or appointment date and time.

7. Cancel an appointment: The Appointments table allows the secretary to search for and delete an existing appointment.

Third: User requirements.

1. Unique and not null primary keys: Each table in the ERD has a primary key property (such as Appointment_ID, Patient_ID, Dentist_ID, etc.) that uniquely identifies each record. The primary keys are defined as non-null and enforce uniqueness, guaranteeing that the tables include no duplicate or missing primary key values.

2. Ensuring foreign key integrity: Using foreign keys, the ERD establishes proper links between tables. The Appointments table, for example, uses foreign keys to refer to the Dentists, Patients, Treatments, Dates, and Times tables (e.g., Dentist_ID, Patient_ID, Treatment_ID, Date, and Time). These foreign keys ensure that the referenced primary keys exist in the relevant tables, preserving referential integrity.

3. Ensuring data validation: Data validation is incorporated into the ERD by specifying data types and lengths for each attribute. In the Patients database, for example, the phone attribute is defined as CHAR(10) , ensuring that it sticks to a predetermined length. The ERD also incorporates constraints like uniqueness, not null, and any additional validations required by the system.

4. Ensuring data persistence: The ERD ensures that appointment data is saved reliably and may be accessed anytime needed using DBMS capabilities. The DBMS ensures that the data is preserved even if the program is terminated.

5. Efficient memory allocation: The ERD's design takes effective memory allocation into account by arranging data into separate tables with relevant properties. This optimizes memory utilization by storing only the essential data in each table, minimizing waste, and enhancing efficiency.

6. Threading techniques for processing large amounts of data: Although the ERD itself does not directly handle processing techniques, the ERD structure and relationships can support the implementation of threading techniques to handle large data sets efficiently.

7. Indexing for fast data retrieval: The ERD facilitates efficient data retrieval by utilizing indexing strategies. Indexing, can be added to attributes such as Dentist_ID, Patient_ID, , allowing for faster data retrieval based on these indexed values. This improves system performance by ensuring quick access to essential appointment information.

8. Interface covering all functionalities: While the ERD defines the database structure rather than the user interface, it can be used to build a user-friendly interface. The tables and relationships in the ERD give a clear structure for providing appointment-related functionalities to the secretary.

Overall, the presented ERD demonstrates a comprehensive and efficient database design that matches with the problem definition, user needs, and system requirements, allowing for successful dental appointment management while resolving the highlighted challenges and constraints.

Criticism of mapping SQL queries to developed CRUD statements & suggested improvements.

Since there are a lot of functions in my DBMS, I will avoid repetitions and only provide key information.

I defined structs for each table (Patient, dentist, date, time, appointment, treatment) and each struct has members with the attributes specified from the table. To store instants of theses structs I used a linked list data structure where every node represents a record of the table and has a pointer that points to the next node (record)

1) SQL Query:

```
SELECT a.Appointment_ID,  p.First_Name, p.Last_Name, d.First_Name, d.Last_Name, t.Treatment_Name, a.Date, a.Time, a.Status

FROM appointments a

JOIN patients p ON a.Patient_ID = p.Patient_ID

JOIN dentists d ON a.Dentist_ID = d.Dentist_ID

JOIN treatments t ON a.Treatment_ID = t.Treatment_ID

WHERE a.Date = CURDATE();
```

CRUD mapping:

To select the fields from the appointment I used current->Attribute. The arrow is used to access the attribute of the structure (the appointment).

To achieve the same functionality of the JOIN statement, I searched for the appointment's Patient ID in the patient's table and returned it to print the first and last name of the patient. Similarly, I followed a similar approach to obtain the first and last name of the dentist and the treatment name from their respective tables, using their respective IDs.

To achieve the WHERE statement in the SQL I used the <time.h> library that includes functions and data types for working with dates and times while the <stdio.h> library is used for standard input and output operations. I

obtained the current date by utilizing the system's local time, and collected the day, month, and year components and stored them in variables which I used to compare to the day, month, and year of the appointment's dates.

2) SQL Query:

```
SELECT *

FROM patients

WHERE Patient_ID = 1;
```

CRUD mapping:

To do the WHERE statement in the C function, I used a binary search to find a specific patient in a linked list of patients. I used a loop that will run until the final pointer is NULL. It identifies the middle element (mid) of the current sublist within the loop. If the sublist is empty, the function returns NULL since the patient with the specified ID could not be found. If the patient ID of the middle element matches the searched patient_id, the method finds the requested patient and provides the mid element. If the middle element's patient ID is smaller than the searched patient_id, the search is narrowed to the second half of the sublist by moving the start pointer to mid->next. If the middle element's patient ID is greater than the specified patient_id, the search focuses on the first half of the sublist.

3) SQL Query:

```
INSERT INTO patients (First_Name, Last_Name, Email , Phone_Number)

VALUES (  'John',  'Doe', 'john@example.com',  '1234567890');
```

CRUD mapping:

Since the SQL function check constraints prior to adding, I replicated the same thing in the implemented C code:

1. I checked to see if the phone number already exists using a while loop, the code iterates through the linked list, comparing the Phone_Number field of each existing patient to the specified phone_number. If a match is detected, the program prints "Phone number already exists" and exits without adding the patient.
2. I Checked to see if the phone number has ten digits by comparing the length of the phone_number string to 10 by using strlen. If the length is less than 10, the function prints "Phone number should be 10 digits" and exits without adding the patient.
3. I checked if the email already exists in a way similar to the phone number check, the code loops through the linked list, comparing each patient's Email field to the provided email; if a match is found, it prints "Email already exists" and exits the function without adding the patient.

To add the new patient and set id correctly (since it is autoincremented),

1. If all of the above checks are successful, a new Patient struct is formed with malloc and the patient's information is copied to the appropriate fields.
2. The next pointer for the new patient is set to NULL.
3. If the linked list is empty , the new patient is assigned Patient_ID = 1 and becomes the head of the linked list.
4. If the linked list is not empty, the function loops through it to identify the last node and generate the highest ID.
5. The new patient's ID is set to id + 1, and is added to the linked list at the end.

4) SQL Query:

```
INSERT INTO patients (First_Name, Last_Name, Email ,Phone_Number)

VALUES ('John', 'Doe', ' ' ,'1234567890');

/*the data filled depends on the user input*/

INSERT INTO dates (day, month, year)

VALUES (20,6,2023);

INSERT INTO times (hour, minute)

VALUES (3,51);
```

INSERT INTO appointments (Patient_ID, Dentist_ID, Treatment_ID, Date, Time, Status)

VALUES (12, 2, 3, 1, 1, 'booked');

CRUD mapping:

These are the steps taken to achieve this in c:

1. It calls the function explained in the previous point to add a new patient.
2. It asks the user to input the appointment details.
3. It creates the date instance using a function that creates a new node and adds it to the dates linked list.
4. It creates the time instance using a function that creates a new node and adds it to the times linked list.
5. It calls a function that creates the appointment:
    1. It initially uses the findPatientByID function to see if the patient with the specified patient_id exists in the patient linked list. If the patient is not found (NULL is returned), the function exits without adding the appointment and shows an error message indicating that the patient does not exist.
    2. If the dentist_id is not -1 (meaning that a specific dentist is assigned), it calls the findDentistByID method to see if the dentist with the specified ID exists in the dentist linked list. If the dentist cannot be found, an error message is displayed and the function is exited.
    3. Similarly, if the treatment_id is not -1 (meaning that a specific treatment is assigned), it calls the findTreatmentByID method to see if the treatment with the specified ID exists in the treatment linked list. If not found, an error message is displayed and the program returns.
    4. The function then examines the linked list to see if an appointment for the specified date, time, and dentist_id already exists. It loops through the linked list, comparing the date, time, and dentist ID of each appointment to the specified values. If a matching appointment is found, an error notice is displayed and the function is exited.
    5. If all of the tests pass, it uses malloc to construct a new struct Appointment instance and assigns the appointment details (patient ID, dentist ID, treatment ID, date, time, and status) from the user.
    6. It then determines where to place the new appointment in the linked list. If the linked list is empty (*head == NULL), the new appointment is assigned as the head and the ID is 1.

Otherwise, it traverses the linked list to identify the last node, increments the new appointment's ID, and sets the last node's next reference to the new appointment.

The other variation of creating a new appointment for a new patient allows the secretary to create an appointment where the dentist, treatment id or both are unknown. Since default is -1, the code sets the unknown value as -1 within the function.

Also, if the user chose to create a new appointment that without creating the new patient, it skips calling the addPatient function.

3) SQL Query:

```
DELETE FROM appointments

WHERE Appointment_ID = 1;
```

CRUD mapping:

1. It initially determines whether the linked list is empty by checking whether *head is NULL. If the list is empty, it returns, quitting the function, and shows a notice explaining that the appointment list is empty.
2. (WHERE statement) It then calls the findAppointmentByID function to try to locate the appointment with the provided ID. If the appointment is not discovered (the returned value is NULL), a message indicating that the appointment with the specified ID was not found is displayed and the function returns.
3. If the deleted appointment is identified, the function determines whether it is the first appointment in the linked list. If this is the case, it moves the head pointer to the next appointment (skipping the appointment to be removed), frees the memory held by the appointment, and displays a message indicating the appointment's successful deletion. The function is then exited by returning.
4. If the appointment to be deleted is not the first, the function searches the linked list for the prior appointment. It stops when it reaches the appointment that comes before the one to be deleted.
5. Once the prior appointment is located, the function adjusts its next pointer to delete the appointment and point to the next appointment. Then it frees the memory held by the to-be-deleted appointment, shows a notice verifying the successful deletion, and exits.

3) SQL Query:

INSERT INTO dates (day, month, year)

VALUES (22,6,2023);

UPDATE appointments

SET Date = 2

WHERE Appointment_ID = 1;

CRUD mapping:

1. The function begins by setting the current appointment pointer to the head of the linked list.
2. If appointmentID is not -1 (meaning that a specific appointment ID is provided), it calls the findAppointmentByID method to try to find the appointment with the specified ID. If the appointment is found (current is not NULL), it updates the appointment's date with the specified data (SET statement),, changes the appointment status to "Rescheduled", shows a message indicating the successful modification, and returns.
3. If appointmentID is -1 and patientID is not -1 (meaning that a specific patient ID is specified), the function enters a loop to look for appointments that are related with the given patient ID. It loops through the linked list, verifying the Patient_ID column for each appointment.
4. If an appointment with a matching Patient_ID is identified, it updates the appointment's date with the specified data (SET statement), displays a message indicating that the modification was successful, and returns.

Suggested improvements

The view appointment SQL simply prints the appointments in the order that they were added to improve it, it can show only the upcoming appointments in ascending order by adding a where and order by clauses.

SELECT * FROM appointments

WHERE Date >= CURDATE()

ORDER BY Date DESC, Time DESC;

For analysis purposes (to see which dentists they should hire more for example) we can add a count function to count the number of appointments for each treatment id.

```
SELECT Treatment_ID, COUNT(*) AS Appointment_Count

FROM appointments

GROUP BY Treatment_ID;
```

Grouping by the patient id when showing the appointments might be a beneficial view for the secretary.

```
SELECT * FROM appointments

GROUP BY Patient_ID;
```

Evaluation of DBMS

The DBMS employs an indexing technique that allows efficient retrieving of data so no matter how much the business expands, and more patients, appointments, treatments, and dentists are added it will still function well. The same way if more records are added the performance of the view function will not be affected as threading technique was employed.

The system's usage of linked lists, with their fast memory management capabilities, is especially important for a growing business. As the company grows and more data is uploaded to the system, such as patients, appointments, treatments, and dentists, linked lists ensure that memory is used properly. So regardless of the increasing volume of data, the system can handle it without affecting performance or memory restrictions significantly.

The provided attributes enclose all the necessary attributes for adding new functionalities in the system. The given attributes include all the attributes required for adding additional functionality to the system. It makes adding new functionalities easier encouraging adaptability, modularity, and reusability, allowing developers to simply expand on current features which is essential for an evolving business.

I suggest that search by doctors by specialty so if a patient wants a doctor with a specific specialty the secretary can search by specialty and assign the wanted doctor. No changes to ERD simply an added function.

I suggest that the secretary can associate the appointment with the dentist’s name instead of id as it may be a daunting task having to see which Id the dentist has. This has its challenges as two dentists can have the same name and the secretary needs to know hoe to spell their name. A drop-down menu would be beneficial in this case. No changes to the ERD.

Instead of relying exclusively on setting the status automatically, the system can also allow manual status editing. This adaptability is especially beneficial when someone is late or there are special conditions that must be considered (ex. no payment made). When the patient approaches the secretary with questions or concerns, having access to complete information on the patient's state allows the secretary to deliver correct and up-to-date comments. To achieve this a new function setStatus should be created that takes the input form the secretary and updates the appointment status.

To address future problems that may arise, it would be beneficial to integrate a payment feature within the system. The system can calculate the amount due to the patient based on the type of treatment received. A new attribute payment due can be added to the patient that keeps track of the money that the patient should pay. This value can also be dependent on other factors that should be considered and added as attributes or tables (structs). This decreases the need of manual calculation of payment due and reduces human error improving the system's efficiency and efficacy benefiting both patients and the business.

Allow the administrator to add and change the details of treatments and dentists to reflect business expansion. No modifications are needed to be done to the ERD but simply newer functions are needed.

It is critical to put in place a backup and recovery system as soon as possible to avoid data loss and to ensure that progress is saved. This solution will protect the data's integrity and reduce the danger of losing crucial information.

References

www.dbvis.com. (n.d.). *A Guide to Multithreading in SQL*. [online] Available at: https://www.dbvis.com/thetable/a-guide-to-multithreading-in-sql/ [Accessed 19 Jun. 2023].
GeeksforGeeks. (2018). *Process-Based and Thread-Based Multitasking*. [online] Available at: https://www.geeksforgeeks.org/process-based-and-thread-based-multitasking/ [Accessed 19 Jun. 2023].
Peterson, R. (2019). *DBMS Concurrency Control: Two Phase, Timestamp, Lock-Based Protocol*. [online] Guru99.com. Available at: https://www.guru99.com/dbms-concurrency-control.html.