

**Database Design and Development**

**10204282**

**H/615/1622**

**Section (3)**

**Designing a Database System**

**Submitted to**

Dr. Raneem Qaddoura

**Submitted on**

April 18th, 2023

**Submitted by**

Marwan Tareq Shafiq Al Farah

**Student ID**

21110011

**Spring 2022 - 2023**

**Table of Content**

[***Report*** 3](#_Toc132734170)

[**Scenario** 3](#_Toc132734171)

[**Data Requirements** 3](#_Toc132734172)

[**User and System Requirements** 5](#_Toc132734173)

[**Conceptual Design** 6](#_Toc132734174)

[**Schema and Mapping** 8](#_Toc132734175)

[**Normalization** 10](#_Toc132734176)

[**Logical Design** 14](#_Toc132734177)

[**Physical Design** 16](#_Toc132734178)

[**The Effectiveness of the Design** 18](#_Toc132734179)

[**Conclusion** 31](#_Toc132734180)

[***References*** 32](#_Toc132734181)

# ***Report***

**Scenario:**

As a junior database developer for an IT firm that provides health-care solutions, I was tasked with creating a relational database model for Ibn Al Haitham Hospital in Amman, Jordan. This reputable healthcare facility provides high-quality medical services to the community, including a wide range of specialties such as cardiology, oncology, orthopedics, and neurology. The hospital has a dedicated operating room (OR) floor that is outfitted with cutting-edge technology and facilities to support surgical procedures in a variety of specialties.

Ibn Al Haitham Hospital’s OR database is to store information about the patients, doctors, ORs, OR beds, and the surgeries. Patient data to be recorded will include name, birth date, gender, phone number, address, age, and allergies. Doctor data that will be stored will include their name, specialty, and contact information. The operating room data will include information such as room number, type, capacity, and equipment. The type of surgery, start and end times, and duration of the surgery will be included in the surgery data. OR bed data will include the size and maximum weight of the bed.

Finally, the database for the OR floor of Ibn Al Haitham Hospital in Amman, Jordan, is an important tool for managing surgical operations and patient information. It enables accurate patient data recording and easy access, resulting in more efficient patient care and management of the hospital’s OR floor.

**Data Requirements:**

1. Each patient is assigned a unique ID, a full name, a birth date, a gender, a phone number, an address, an age, and a list of allergies.
   * **Unique identifier:** a unique identifier for each patient that is required for easy identification and management of patient records.
   * **Name:** To facilitate searching, sorting, and organizing patient records, the name should be stored separately for each component (first, middle, and last).
   * **Birth Date:** Needed to calculate the patient’s age and for various medical purposes.
   * **Gender:** This is critical for identifying patients and ensuring appropriate treatment plans.
   * **Phone Number:** Required for communication and emergency contact.
   * **Address:** This information is required for patient identification, billing, and geographical analysis.
   * **Age:** Aids in the selection of appropriate treatment and medical care for patients.
   * **Allergies:** A list of allergies for each patient (may include more than one allergy at a time).
2. Each doctor is identified by a unique ID, full name, specialty, and phone number.
   * **Unique ID:** Each doctor has a unique identifier, which is required for easy identification and management of doctor records.
   * **Name:** To allow for easier searching, sorting, and organization of doctor records, the name should be stored separately for each component (first, middle, and last).
   * **Specialty:** A doctor’s area of expertise, which is important for scheduling surgeries and assigning patients.
   * **Phone Number:** Required for communication and emergency contact.
3. Each operating room has its own ID, room number, room type, capacity, and equipment list.
   * **Unique ID:** Each operating room requires a unique identifier for easy identification and management of operating room records.
   * **Room Number:** A physical location or designation within the facility that is required for surgery scheduling.
   * **Room Type:** Classifies operating rooms according to their purpose and usage, which aids in scheduling and resource allocation.
   * **Capacity:** The number of staff that an operating room can accommodate, which is useful for planning and organization.
   * **Equipment:** A list of the equipment in each operating room (which may contain more than one piece of equipment at the same time).
4. Each surgery has its own ID, type, duration, start and end times.
   * **Unique ID:** Each surgery requires a unique identifier for easy identification and management of surgery records.
   * **Surgery Type:** Describes the procedure being performed and is required for scheduling and documentation.
   * **Start and End Times:** Specify the time frame for the surgery, which is required for scheduling, planning, and resource management.
   * **Operation Duration:** Aids in planning and resource allocation.
5. Each OR bed has a size and a maximum weight capacity.
   * **Bed Size:** The dimensions of the bed, which are important for patient comfort and operating room fit.
   * **Max Weight:** This setting specifies the maximum weight that the bed can support, which is required for patient safety.
6. A doctor can perform multiple surgeries, but each surgery is performed by only one doctor.
7. A surgery can involve only one patient, and a patient can have multiple surgeries.
8. An operating room can host multiple surgeries, but each surgery is performed in only one operating room.
9. Each OR is assigned to one OR bed, and each OR bed is assigned to only one OR.
10. Each doctor has multiple patients, and each patient can have multiple doctors. The database keeps track of the appointment dates and times between doctors and patients.

**User and System Requirements:**

* + 1. **Patients:**

Patients should be able to:

* + View, add, or change their personal information, including contact information (phone number) and address.
  + View their personal information, which includes their name, birthdate, age, gender, allergies, and identification.
  + View information about upcoming surgeries, such as surgery details and the assigned operating room.
  + View information about the doctors who are involved in their care, including their names and specialties.
  + View available appointment slots with their assigned doctors and schedule appointments around them.
  + View, add, modify, or delete their scheduled appointments.
    1. **Doctors:**

Doctors should be able to:

* + View, add, or modify their personal information, such as contact information (phone number).
  + View their personal information, such as names, specialties, and ID.
  + View, add, or delete patients under their care, including demographics, contact information, and medical history.
  + View, add, or change upcoming surgery information for patients under their care, including surgery details and operating room assignment.
  + View operating room information, such as the equipment available in each room.
  + Check out their appointment schedules, which include patient names, contact information, and appointment times.
  + Set their appointment availability, which includes adjusting time slots and blocking out time for breaks, meetings, or other activities.
  + View their patients’ appointment history.
    1. **Hospital Administrators:**

Hospital administrators should be able to:

* + View and update patient information, including demographics, contact information, medical history, and allergies.
  + Add, update, or delete doctor information, such as names, specialties, and contact information.
  + View, add, update, or delete all patients’ surgery information, including surgery details, assigned doctors, and operating rooms.
  + View, add, update, or delete operating room data, such as room IDs and equipment.
  + View, add, update, or delete operating room beds data, such as their sizes or maximum weights.
  + Assign specific surgeons to specific surgeries.
    1. **System Requirements:**

The medical database system should be designed to meet the following requirements:

* + Create conceptual, logical, and physical designs for the database, along with providing the schema of the database, and performing mapping for the attributes and relationships along with normalization up to the third normal form.
  + Proper functionality design, to ensure data completeness, consistency, and integrity.
  + Improve system performance by providing efficient query and storage performance.
  + Ensure modularity, flexibility, scalability, and proper documentation to aid system maintenance and to accommodate an increasing number of users.
  + Be secure and in accordance with healthcare regulations such as HIPAA, which include patient privacy and data security.
  + Provide a simple interface to ensure user-friendliness.
  + Integrate with other hospital-related systems like laboratory, radiology, billing, and electronic health records to allow for seamless data exchange.

**Conceptual Design:** (MariaDB, 2016; Thiru, 2016; Informatica e Ingegneria Online, 2020)

The first phase of database design is conceptual design, in which an abstract representation of the database is created to identify and describe entities and relationships that reflect the organization’s data requirements. To create an effective conceptual design, the designer must have a thorough understanding of the organization’s data requirements. This phase lays the groundwork for the subsequent stages of database design, which include logical and physical design.

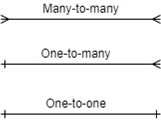
The two main components of conceptual design are entities and relationships. Entities are the objects or concepts about which the organization wishes to store data about, whereas relationships are the associations between entities. An Entity-Relationship Diagram (ERD) is used to represent the overall structure of a database system, including its entities, attributes, and relationships. An ERD is a critical tool during the conceptual design stage, assisting in ensuring that the design meets the business requirements and accurately reflects the information that must be stored in the database.

Crow’s Foot Notation was used in the design of the database, and it is a widely used visual language for creating Entity-Relationship Diagrams (ERDs) during the database’s design phase, making it easier to understand the system’s relationships and entities by representing cardinalities and cardinality ratios. It uses the following symbols to represent entities and relationships: (Dybka, 2016; Abba, 2022)

* + **Entities:** are represented as rectangles with the entity name inside.



* + **Relationships:** Represent associations or connections between entities, such as a patient-doctor relationship indicating that a patient has been assigned to a specific doctor.
  + **Cardinality:** The maximum and minimum number of instances of one entity that can be related to instances of another. One doctor, for example, can be responsible for multiple patients, whereas a patient can only have one primary doctor.



Once completed, the conceptual design serves as a roadmap for the subsequent stages of database design and helps to ensure that the final database meets the requirements of the organization. By identifying potential design issues early in the development process, changes can be made before moving on to the logical and physical designs, and before implementation.

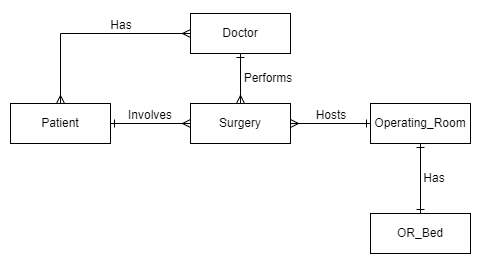


Figure 1 – The database’s conceptual design

The conceptual design presented above was built based on the following information that was extracted from the data requirements:

1. The five primary entities in the database are Patient, Doctor, Operating\_Room, Surgery, and OR\_Bed.
2. The following lists these entities’ individual cardinalities as well as the cardinality ratios between them:

* The relationship between Doctor (one) and Surgery (many) is one-to-many.
* The relationship between Surgery (many) and Patient (one) is one-to-many.
* The relationship between Operating\_Room (one) and Surgery (many) is one-to-many.
* The relationship between Operating\_Room (one) and OR\_Bed (one) is one-to-one.
* The relationship between Doctor (many) and Patient (many) is many-to-many.

Therefore, the conceptual design that is presented in Figure 1 uses Crow’s Foot Notation to represent visually these entities and their cardinality ratios (relationships) within the database.

**Schema and Mapping:**

A database schema, which consists of tables, columns, primary keys (PK), foreign keys (FK), and other components, is a blueprint that specifies the structure, relationships, and constraints of a database. Primary keys serve as distinct identifiers for each row in a table, preserving data accuracy and facilitating quick retrieval. Tables are connected by means of foreign keys, which connect a column in one table to the primary key in another. This promotes referential integrity and eliminates inconsistencies. (Lucidchart, 2017; IBM, 2019)

Mapping in database design is the process of converting actual ideas into a logical structure. It involves dealing with derived attributes, composite attributes, multivalued attributes, and relationships between entities. The database schema is made to be well-organized, effective, and optimized through this process. (Informatica, 2019; Keating, 2021; Talend, 2022)

**Note:** whenever the schema is shown, when an attribute is underlined, it means that this attribute is a primary key, and when an attribute is highlighted in yellow it means that this attribute is a foreign key.

**Schema before mapping and before determining the keys:** Based on the data requirements above, it was concluded that the initial schema of the database is as follows:

1. Patient (patient\_id, full\_name, birth\_date, gender, phone\_number, address, age, allergies)
2. Doctor (doctor\_id, full\_name, specialty, phone\_number)
3. Operating\_Room (operating\_room\_id, room\_number, room\_type, capacity, equipment)
4. Surgery (surgery\_id, surgery\_type, surgery\_duration, start\_time, end\_time)
5. OR\_Bed (bed\_size, max\_weight)

Below we will go over all the candidate keys, primary keys, prime attributes, and non-prime attributes of the entities below. The primary key is the selected unique identifier for records in a table, whereas candidate keys are potential unique identifiers for records in a table. Non-prime attributes do not help with record unique identification, whereas prime attributes are a component of candidate keys and play a role in unique identification.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entity | Candidate Keys | Primary Key | Prime Attributes | Non-Prime Attributes |
| Patient | patient\_id, phone\_number | patient\_id | patient\_id, phone\_number | full\_name, birth\_date, gender, address, age, allergies |
| Doctor | doctor\_id, phone\_number | doctor\_id | doctor\_id, phone\_number | full\_name, specialty |
| Operating\_Room | operating\_room\_id, room\_number | operating\_room\_id | operating\_room\_id, room\_number | room\_type, capacity, equipment |
| Surgery | surgery\_id | surgery\_id | surgery\_id | surgery\_type, surgery\_duration, start\_time, end\_time |
| OR\_Bed | - | - | - | bed\_size, max\_weight |

It is crucial to note that the “The Effectiveness of the Database Design” section will go into great detail on the justification for choosing each attribute as a candidate, primary, prime, or non-prime attribute. The choices made during the database design process will be clearly understood after reading this explanation, along with how they affect the schema’s overall effectiveness and efficiency.

**Schema before mapping and after determining the keys:**

1. Patient (patient\_id, full\_name, birth\_date, gender, phone\_number, address, age, allergies)
2. Doctor (doctor\_id, full\_name, specialty, phone\_number)
3. Operating\_Room (operating\_room\_id, room\_number, room\_type, capacity, equipment)
4. Surgery (surgery\_id, surgery\_type, surgery\_duration, start\_time, end\_time)
5. OR\_Bed (operating\_room\_id, bed\_size, max\_weight)

An updated schema that more accurately depicts the concepts and relationships found in the real world is created after the mapping process is complete. The “The Effectiveness of the Database Design” section will go into great detail about the specific steps of the mapping process, which include handling derived attributes, composite attributes, multivalued attributes, and creating relationships between entities. This justification will offer insightful information about how the mapping process enhanced the original schema and made it better suited for accurate and efficient data storage and retrieval.

**Schema after mapping and after determining the keys:**

1. Patient (patient\_id, first\_name, middle\_name, last\_name, birth\_date, gender, phone\_number, address)
2. Doctor (doctor\_id, first\_name, middle\_name, last\_name, specialty, phone\_number)
3. Operating\_Room (operating\_room\_id, room\_number, room\_type, capacity)
4. Surgery (surgery\_id, surgery\_type, start\_time, end\_time, doctor\_id, patient\_id, operating\_room\_id)
5. OR\_Bed (operating\_room\_id, bed\_size, max\_weight)
6. Patient\_Allergies (patient\_id, allergy)
7. Doctor\_Patient (doctor\_id, patient\_id, appointment\_date, appointment\_time)
8. OR\_Equipment (operating\_room\_id, equipment)

**Normalization:**

Normalization is the process of structuring and optimizing data in a database for effective querying and maintenance. It entails dividing a sizable table into smaller, more manageable tables connected to one another by a clear set of rules. Eliminating data redundancy, reducing data update anomalies, and ensuring data consistency and integrity are the objectives of normalization. A set of normalization guidelines, referred to as normal forms, are applied to the database schema during the procedure. Each of the normal forms, which range from the first normal form (1NF) to 5NF or 6NF, builds on the one before it to further optimize the database’s structure. We will only apply normalization to the third normal form (3NF) in this database design. (Chris, 2021; Javatpoint, 2021; Peterson, 2022; Qiu *et al.*, 2022)

1. **1st Normal Form (1NF):** (StudyTonight, 2020; Javatpoint, 2021; Peterson, 2022)

1NF is a database normalization form that ensures each attribute of a relation is atomic, which means that each attribute can only hold one value. A relation is said to be in 1NF if all of its attributes contain only atomic values, implying that each attribute is a simple attribute (not composite nor multivalued).

When an attribute in a relation contains non-atomic values, such as a set of values, lists, or other complex data types, 1NF is violated. To achieve 1NF, these non-atomic attributes must be decomposed into separate attributes or relations, which can then be linked using foreign keys.

Composite attributes are in violation of 1NF because they have multiple sub-attributes. This makes it challenging to carry out specific database operations like sorting, searching, and data querying. In order to adhere to the 1NF, composite attributes must be divided into individual attributes, with each attribute containing a single piece of data.

Due to the possibility of having multiple values for a single attribute, multivalued attributes go against 1NF. Data redundancy and inconsistency may result from this, making the database challenging to manage and maintain. For each multivalued attribute that has more than one value, a new relation must be created in order to comply with the 1NF and ensure that each attribute only has one value.

Since we addressed these issues during the mapping process, our current schema is in 1st Normal Form (1NF) which mean that we have already made all of the attributes atomic in the given schema by handling composite and multivalued attributes. The table below summarizes the relationships, attributes, violation descriptions, and solutions:

|  |  |  |  |
| --- | --- | --- | --- |
| Relations | Attributes | Violation description | Solution – Relations |
| Patient (patient\_id, full\_name, birth\_date, gender, phone\_number, address) | full\_name | Composite attribute (description provided above) | Patient (patient\_id, first\_name, middle\_name, last\_name, birth\_date, gender, phone\_number, address) |
| Doctor (doctor\_id, full\_name, specialty, phone\_number) | full\_name | Doctor (doctor\_id, first\_name, middle\_name, last\_name, specialty, phone\_number) |
| Patient (patient\_id, first\_name, middle\_name, last\_name, birth\_date, gender, phone\_number, address, allergies) | allergies | Multivalued attribute (description provided above) | Patient (patient\_id, first\_name, middle\_name, last\_name, birth\_date, gender, phone\_number, address)  Patient\_Allergies (patient\_id, allergy) |
| Operating\_Room (operating\_room\_id, room\_number, room\_type, capacity, equipment) | equipment | Operating\_Room (operating\_room\_id, room\_number, room\_type, capacity)  OR\_Equipment (operating\_room\_id, equipment) |

1. **2nd Normal Form (2NF):** (StudyTonight, 2020; Javatpoint, 2021; Qiu *et al.*, 2022)

By ensuring that every non-prime attribute in a relation is fully functionally dependent on the entire primary key, it is a database normalization form that improves upon 1NF. If a relation satisfies the following requirements, it is said to be in 2NF:

* 1. The relationship is in the first normal form (1NF) already.
  2. Each non-prime attribute is fully functionally dependent on the entire primary key.

When a relation only partially depends on the primary key in the case of a composite primary key, this is known as partial functional dependency and constitutes a violation of 2NF. These partial dependencies should be eliminated in order to achieve 2NF by breaking the relation up into smaller relations that can be linked using foreign keys.

This is accomplished by locating the non-prime attributes that are dependent on just a portion of the primary key and establishing a new relation for them. Only the attributes that are dependent on the new relation’s primary key, which is the component of the primary key on which they were previously dependent, will be included in the new relation.

We can link the relations using foreign keys after they have been broken down. This will create a connection between the old and new relations, enabling us to maintain the integrity of the data and guarantee that every attribute is fully dependent on the primary key. This method allows us to achieve the 2NF and resolve partial functional dependencies.

Our current schema is already in 1NF, and when we checked to see if it was also in 2NF, we found that there were no 2NF violations. This means that our schema is in 2NF because all non-prime attributes are completely functionally dependent on their primary keys. To show a 2NF violation and how to fix it, let us construct the following hypothetical scenario:

**Hypothetical Scenario:**

Suppose we have a relation “Doctor\_Schedule” that stores information about the doctors, and their work schedules.

|  |  |  |  |
| --- | --- | --- | --- |
| Relations | FDs | Violation description | Solution – Relations |
| Doctor\_Schedule (doctor\_id, day, schedule, first\_name, middle\_name, last\_name, specialty, phone\_number) | doctor\_id, day → schedule, first\_name, middle\_name, last\_name, specialty, phone\_number | Partial dependency (description provided above) | A new entity called Doctor (doctor\_id, first\_name, middle\_name, last\_name, specialty, phone\_number)  A new entity called Schedule (doctor\_id, day, schedule) |
| doctor\_id → first\_name, middle\_name, last\_name, specialty, phone\_number |

The “Doctor\_Schedule” relation in this hypothetical scenario has a composite primary key made up of “doctor\_id” and “day”. The ‘first\_name’,’middle\_name’, ‘last\_name’,’specialty’, and ‘phone\_number’ attributesare only functionally dependent on the ‘doctor\_id’ attribute, not the entire primary key, resulting in a partial functional dependency. We divided the “Doctor\_Schedule” relation into two relations: “Doctor” and “Schedule” in order to solve this problem and achieve 2NF. Each non-prime attribute is now fully functionally dependent on the primary key of its specific relation.

1. **3rd Normal Form (3NF):** (StudyTonight, 2020; Javatpoint, 2021; Peterson, 2022)

It is a database normalization form that improves on 2NF by guaranteeing that none of a relation’s non-prime attributes are transitively dependent on the primary key. If a relation satisfies the requirements listed below, it is said to be in 3NF:

* 1. The relationship is in second normal form (2NF) already.
  2. Non-prime attributes do not depend on each other transitively.

When a relation contains transitive dependencies, which occur when one non-prime attribute depends on another non-prime attribute, which in turn depends on the primary key, 3NF is violated. Due to the redundancy and data anomalies it causes, this scenario breaks the 3NF normalization rule. These transitive dependencies should be eliminated by breaking the relation up into smaller relations in order to achieve 3NF.

To solve this problem, we must divide the relationship into two separate relations. We can create a new relation that only includes the dependent and primary key attributes. Then, we create a second relation that includes the transitive dependency’s non-prime attribute and the primary key attribute.

The primary key of the first relation may be referred to by the second relation as a foreign key, and the first relation may serve as the primary relation. By doing this, we get rid of the transitive dependency and create a design that complies with 3NF. With this method, data redundancy is reduced, data consistency is enhanced, and update anomalies are prevented in the database.

Our current schema is already in 2NF, and after checking to see if it is also in 3NF, we discovered that there are no 3NF violations in our current schema, indicating that our schema is in 3NF. Non-prime attributes are all non-transitive dependencies on their primary keys. However, let us create a hypothetical scenario to demonstrate a 3NF violation and its solution:

**Hypothetical Scenario:**

Suppose we have a relation “Patient\_Info” that stores information about patients, their insurance companies, and the corresponding insurance company’s addresses.

**Schema:** Patient (patient\_id, first\_name, middle\_name, last\_name, birth\_date, gender, phone\_number, address, insurance\_company, insurance\_address)

|  |  |  |  |
| --- | --- | --- | --- |
| Relations | FDs | Violation description | Solution – Relations |
| Patient (patient\_id, first\_name, middle\_name, last\_name, birth\_date, gender, phone\_number, address, insurance\_company, insurance\_address) | patient\_id → insurance\_company | Transitive dependency (description provided above) | A new entity called Patient (patient\_id, first\_name, middle\_name, last\_name, birth\_date, gender, phone\_number, address, insurance\_company)  A new entity called Insurance (insurance\_company, insurance\_address) |
| insurance\_company → insurance\_address |
| patient\_id → insurance\_address |

The “Patient\_Info” relation in this hypothetical scenario has the primary key ‘patient\_id’. The ‘insurance\_address’ attribute is transitively dependent on the ‘patient\_id’ attribute via the ‘insurance\_company’ attribute. To solve this problem and achieve 3NF, we split the “Patient\_Info” relation into two relations: “Patient” and “Insurance”. As a result, each non-prime attribute is no longer transitively dependent on its respective relation’s primary key.

**Logical Design:** (GeeksforGeeks, 2011; Broadcom, 2013; Borgida, Casanova and Laender, 2016; Informatica e Ingegneria Online, 2020)

Logical design is a stage in database design that focuses on organizing and structuring data elements to accurately represent business requirements and data relationships.

The main goal of the logical design phase of database development is to create a clear and well-structured representation of an organization’s data requirements, without focusing on how the data will be physically stored. It entails defining entities, attributes, and relationships between entities to enable efficient querying and data management. The conceptual design, which is a high-level, abstract representation of the organization’s data requirements, is used to create the logical design. In other words, the logical design is a visual representation of the schema.

A logical design includes the following elements:

* **Entities:** In the context of a hospital, entities represent real-world objects such as patients, doctors, and operating rooms.
* **Attributes:** These are the characteristics or properties of an entity, such as the Patient entity’s patient\_id, first\_name, or birth\_date.
* **Relationships:** Represent associations or connections between entities, such as a patient-doctor relationship indicating that a patient has been assigned to a specific doctor.
* **Cardinality:** The maximum and minimum number of instances of one entity that can be related to instances of another. One doctor, for example, can be responsible for multiple patients, whereas a patient can only have one primary doctor.

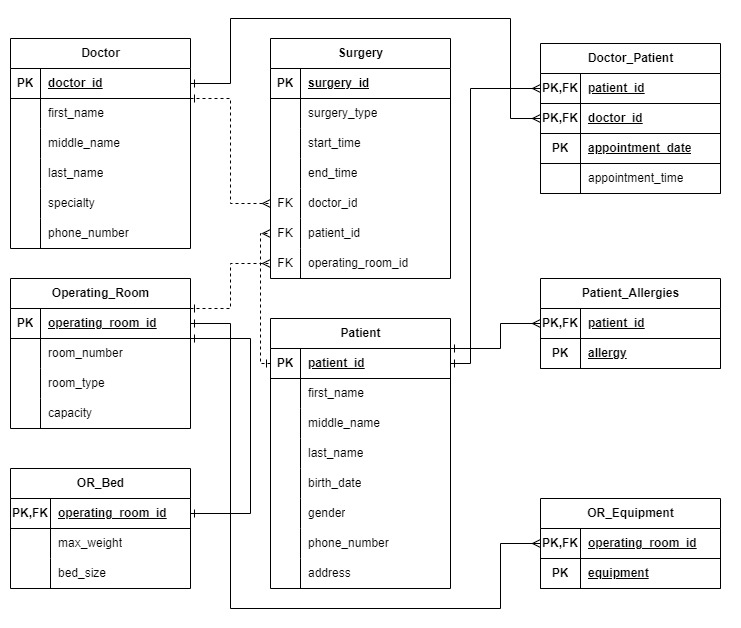


Figure 2 – The database’s logical design

The diagram above depicts the logical design of our hospital database, with entities depicted as rectangles, attributes listed within the rectangles, and relationships between entities depicted as lines connecting the rectangles. Symbols on both ends of the lines indicate the cardinality of the relationships (cardinality ratios). It also represents the weak relationships between the entities as dotted lines while representing the strong relationships between entities as solid lines (Watt, 2014). Additionally, next to each entity we can specify whether it was PK, FK, both, or neither.

The logical design does not add any new information to the database design, as it is merely a visual representation of the schema after mapping:

1. Patient (patient\_id, first\_name, middle\_name, last\_name, birth\_date, gender, phone\_number, address)
2. Doctor (doctor\_id, first\_name, middle\_name, last\_name, specialty, phone\_number)
3. Operating\_Room (operating\_room\_id, room\_number, room\_type, capacity)
4. Surgery (surgery\_id, surgery\_type, start\_time, end\_time, doctor\_id, patient\_id, operating\_room\_id)
5. OR\_Bed (operating\_room\_id, bed\_size, max\_weight)
6. Patient\_Allergies (patient\_id, allergy)
7. Doctor\_Patient (doctor\_id, patient\_id, appointment\_date, appointment\_time)
8. OR\_Equipment (operating\_room\_id, equipment)

**Physical Design:** (IBM, 2017; Informatica e Ingegneria Online, 2020; Banister, 2021; Chola, 2022)

The actual implementation and storage of data on the system are the focus of the physical design stage of database design. By selecting the proper storage structures, indexing strategies, and data access techniques, the physical design phase aims to optimize the database’s performance, storage, and data retrieval. The logical design is converted during this stage into a format that the database management system (DBMS), in our case, MySQL, can use.

The components of the physical design are the same as the logical design, however, in the physical design we determine the datatypes for each attribute along with any additional constraints that we want to add:

* **Data types:** The types of data that can be stored in each attribute.
* **Constraints:** These make sure that the database’s data is compliant with established guidelines or requirements.

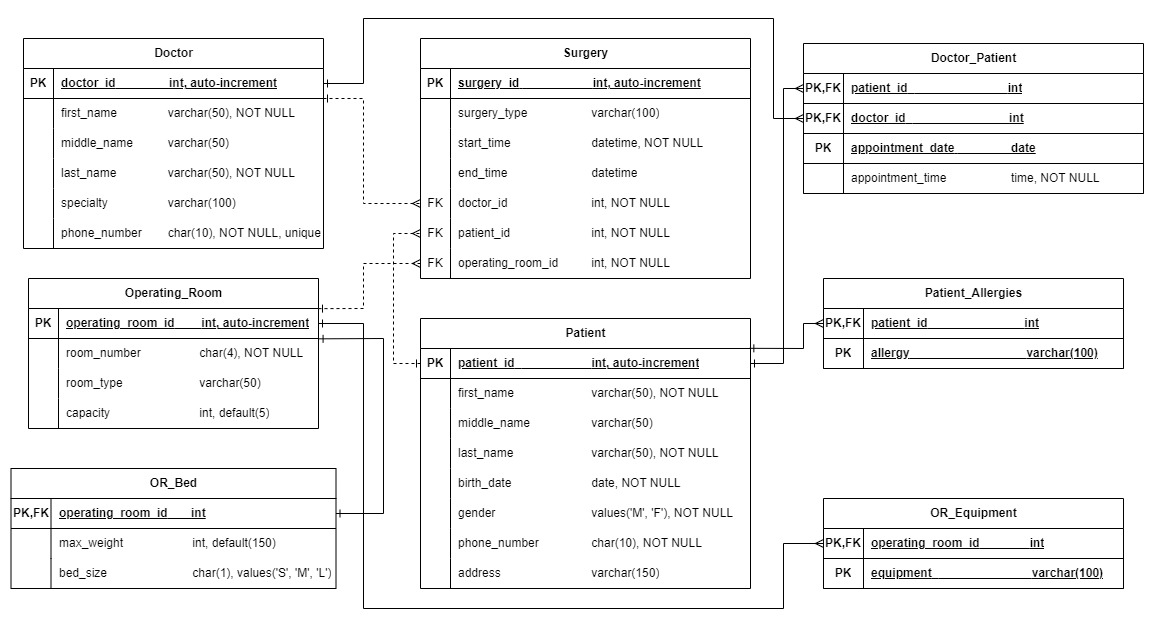


Figure 3 – The database’s physical design

The diagram above depicts the physical design of our hospital database, including the tables, columns, data types, and constraints. The physical design is a visual representation of the schema listed below:

1. **Patient**

* patient\_id [int, auto-increment,PK]
* first\_name [varchar(50), NOT NULL]
* middle\_name[varchar(50)]
* last\_name [varchar(50), NOT NULL]
* birth\_date [date, NOT NULL]
* gender [values(‘M’, ‘F’), NOT NULL]
* phone\_number [char(10), NOT NULL]
* address [varchar(150)]

1. **Doctor**

* doctor\_id [int, auto-increment, PK]
* first\_name [varchar(50), NOT NULL]
* middle\_name[varchar(50)]
* last\_name [varchar(50), NOT NULL]
* specialty [varchar(100)]
* phone\_number [char(10), NOT NULL, unique]

1. **Operating\_Room**

* operating\_room\_id [int, auto-increment, PK]
* room\_number [char(4), NOT NULL]
* room\_type [varchar(50)]
* capacity [int, default(5)]

1. **Surgery**

* surgery\_id [int, auto-increment, PK]
* surgery\_type [varchar(100)]
* start\_time [datetime, NOT NULL]
* end\_time [datetime]
* doctor\_id [int, NOT NULL, FK]
* patient\_id [int, NOT NULL, FK]
* operating\_room\_id [int, NOT NULL, FK]

1. **OR\_Bed**

* bed\_size [char(1), values(‘S’, ‘M’, ‘L’)]
* maximum\_weight [int, default(150)]
* operating\_room\_id [int, PK, FK]

1. **Patient\_Allergies**

* patient\_id [int, PK, FK]
* allergy [varchar(100), PK]

1. **OR\_Equipment**

* operating\_room\_id [int, PK, FK]
* equipment [varchar(100), PK]

1. **Doctor\_Patient**

* patient\_id [int, PK, FK]
* doctor\_id [int, FK, PK]
* appointment\_date (PK) [date, PK]
* appointment\_time [time]

Data types are chosen in the physical design of the hospital database to effectively store attributes and enhance storage and query performance. The chosen data types are as follows: (ORACLE, 2022)

* **int:** stores whole numbers such as IDs and capacities. MySQL’s int supports a wide range, meeting the database’s requirements.
* **varchar:** Variable-length character strings, such as names and textual attributes, are stored in varchar. Maximum lengths are used to achieve efficient storage.
* **char:** Used to store fixed-length strings such as phone\_number and room\_number. With specified values (e.g., char(10), char(4)), consistent lengths are ensured.
* **date and time:** Separately store date and time values for efficient storage and easy querying/comparison.
* **datetime:** Combines date and time types, making it easier to store and retrieve combined values such as surgery start and end times.

Several constraints are used to ensure the consistency, accuracy, and integrity of data, and these include:

* **NOT NULL:** Ensures that every record has a value. Prevents inconsistency by ensuring complete and accurate information for essential attributes.
* **UNIQUE:** Requires that all values in a column be unique. No two records have the same value, which ensures accuracy and avoids confusion/errors.
* **DEFAULT:** When no value is specified, it is used to assign default values. Ensures that certain attributes have consistent values during data entry.
* **AUTO\_INCREMENT:** This is used for primary key columns. Maintains data consistency and integrity by assigning unique, automatically incremented integer values.
* **VALUES:** A customized constraint that limits input to predefined values. Prevents incorrect data entry and ensures data consistency.

These data types and constraints provide efficient and accurate data representation, while also maintaining table relationships and optimizing performance.

**The Effectiveness of the Design:**

* **Entities and Relationships:**

Entity and relationship choices are critical in database design. In this section, we will explain why each entity and relationship was chosen for the database, as well as what would happen if another one was chosen, an entity was not chosen at all, or two entities were accidentally combined into a single entity.

The database's five primary entities are Patient, Doctor, Operating\_Room, Surgery, and OR\_Bed.

Each of these entities was selected because they represent critical aspects of hospital operations. The Patient entity is required to keep track of patient data, such as personal information and medical history. The Doctor entity is required to manage physician data such as specialty and availability. The Surgery entity stores information about each procedure, while the Operating\_Room entity is critical for scheduling surgeries. Finally, the OR\_Bed entity is critical for keeping patients comfortable and safe during surgery.

If any of these entities were not selected, or if another entity was selected in their place, the database would not accurately reflect the hospital's operations. For example, if the Patient entity was not selected, keeping track of patient information would be difficult. Similarly, if Medication was used instead of OR\_Bed, it would not reflect the physical aspects of surgical procedures, which are critical for patient safety, and would not have met the hospital's data requirements.

If two entities are mistakenly combined into a single entity, data redundancy and inconsistency may result. For example, if the Patient and Doctor entities were combined into a single entity, each record would contain information about both the patient and the doctor. Because each patient may have multiple doctors, and each doctor may have multiple patients, this could result in duplicated data. Inconsistencies in data entry and management may also result, as some patients may have incomplete doctor information or vice versa.

The following shows the individual cardinalities of these entities as well as the cardinality ratios between them:

* + The relationship between Doctor (one) and Surgery (many) is one-to-many.
  + The relationship between Surgery (many) and Patient (one) is one-to-many.
  + The relationship between Operating\_Room (one) and Surgery (many) is one-to-many.
  + The relationship between Operating\_Room (one) and OR\_Bed (one) is one-to-one.
  + The relationship between Doctor (many) and Patient (many) is many-to-many.

These cardinality ratios were chosen to accurately reflect each entity's relationships. For example, because Doctor and Surgery have a one-to-many relationship, a single doctor can perform multiple surgeries, but each surgery is performed by only one doctor, and so on.

If different cardinality ratios were chosen for any of these relationships, the database's representation of the hospital's operations would be less accurate. For example, if the relationship between Doctor and Surgery was many-to-many, a single surgery could be performed by multiple doctors, which is not accurate in most hospital settings. Similarly, a many-to-one relationship between Operating\_Room and Surgery would imply that a surgery can be hosted in multiple operating rooms at the same time, but each operating room can only host one surgery.

* **Candidate and Primary keys, and Prime and Non-Prime attributes:**
  + - 1. **Patient:**
* Candidate keys are chosen because they can identify a tuple in the relation uniquely. Patient\_id and phone\_number are considered candidate keys in this case because they can both uniquely identify a patient.
* The primary key patient\_id was chosen because it is a dedicated unique identifier for patients, whereas phone\_number may change or be shared in certain cases (e.g., multiple family members using the same number).
* Non-prime attributes such as first\_name, middle\_name, last\_name, birth\_date, gender, address, and age are unsuitable for use as candidate keys because they cannot uniquely identify a patient.
  + - 1. **Doctor:**
* Because doctor\_id and phone\_number can both uniquely identify a doctor, they are both considered candidate keys.
* The primary key was chosen as doctor\_id because it is a unique identifier created specifically for doctors, whereas phone\_number can change.
* Non-prime attributes such as first\_name, middle\_name, last\_name, and specialty are unsuitable as candidate keys because they cannot uniquely identify a doctor.
  + - 1. **Operating\_Room:**
* The candidate keys operating\_room\_id and room\_number were chosen because they can uniquely identify an operating room.
* The primary key operating\_room\_id was chosen because it is a dedicated unique identifier, whereas room\_number could change (e.g., room renumbering or duplication in different buildings).
* Non-prime attributes such as room\_type, capacity, and equipment are unsuitable as candidate keys because they cannot uniquely identify an operating room.
  + - 1. **Surgery:**
* The candidate key is surgery\_id because it can uniquely identify a surgery.
* Because surgery\_id is a dedicated unique identifier for surgeries, it was chosen as the primary key.
* Non-prime attributes such as surgery\_type, surgery\_duration, start\_time, and end\_time are unsuitable as candidate keys because they cannot uniquely identify a surgery alone or in combination.
  + - 1. **OR\_Bed:**
* The OR\_Bed entity is a weak entity because no candidate keys are provided in the schema. To form a complete relation, it must be linked to another entity via a foreign key.
* Non-prime attributes such as bed\_size and max\_weight are unsuitable as candidate keys because they cannot uniquely identify an OR\_Bed.
* To address this weak entity, the foreign key "operating\_room\_id" was added to the OR\_Bed entity. The choice of "operating\_room\_id" as the foreign key establishes a clear relationship between the OR\_Bed and Operating\_Room entities. Each OR\_Bed is assigned to a specific operating room, and this association aids in uniquely identifying an OR\_Bed in the context of its corresponding Operating\_Room.
  + - 1. **Patient\_Allergies:**
* This entity's candidate key is a combination of patient\_id and allergy. Because a patient can have multiple allergies and each allergy can be linked to multiple patients, both columns are required to uniquely identify a patient allergy combination.
* This entity's primary key is the same as the candidate key, which is the combination of patient\_id and allergy.
  + - 1. **OR\_Equipment:**
* This entity's candidate key is a combination of operating\_room\_id and equipment. Because each equipment item in an operating room can be associated with multiple operating rooms, both columns are required to uniquely identify an equipment item in an operating room.
* This entity's primary key is the same as the candidate key, which is the combination of operating\_room\_id and equipment.
  + - 1. **Doctor\_Patient:**
* This entity's candidate key is a combination of patient\_id, doctor\_id, and appointment\_date. Each patient may have multiple appointments with multiple doctors on different dates, as may each doctor. As a result, all three columns must be present in order to uniquely identify a patient-doctor-appointment combination.
* This entity's primary key is the same as the candidate key, which is a combination of patient\_id, doctor\_id, and appointment\_date.
* Non-prime attributes, such as appointment\_time, are unsuitable as candidate keys because they cannot uniquely identify a patient-doctor-appointment combination alone or in combination.
* **The Mapping Process:**

The mapping process was carried out in order to create a well-structured, efficient, and accurate representation of the hospital database while adhering to database design principles. This section discusses the reasons for categorizing attributes as composite, derived, or multivalued, as well as the methods used to handle these attributes and the reasoning behind these decisions, using examples from the original database and hypothetical scenarios.

1. **Classification of the Attributes:**

Based on their inherent properties, attributes were classified as composite, derived, or multivalued. Composite attributes are made up of several components, such as 'full\_name' in the Patient and Doctor entities, which is made up of first, middle, and last names. Derived attributes, such as 'age' in the Patient entity and 'surgery\_duration' in the Surgery entity, can be calculated from other attributes, such as the 'birth\_date' and the difference between the 'end\_time' and 'start\_time', respectively. Multiple values for a single entity instance can be assigned to multivalued attributes, such as multiple 'allergies' for a patient or multiple 'equipment' in an Operating\_Room.

1. **Handling of the Attributes:**

To create an organized and efficient schema, each attribute type was handled based on its classification:

* Composite attributes were separated into individual attributes to facilitate data searching and organization. In the Patient and Doctor entities, for example, the 'full\_name' attribute was divided into 'first\_name, ‘middle\_name’, and 'last\_name'.
* Derived attributes were removed to avoid redundancy and inconsistency by calculating these values from existing data rather than directly storing them. The 'age' attribute in the Patient entity, for example, and the 'surgery\_duration' attribute in the Surgery entity were both removed.
* To address multivalued attributes, new entities were created, ensuring a normalized database structure and efficient querying. To handle the 'allergies' attribute in the Patient entity and the 'equipment' attribute in the Operating\_Room entity, for example, new entities 'Patient\_Allergies' and 'OR\_Equipment' were created.

1. **Best Solution Rationale:**

The following reasons led to the selection of the best methods for handling attributes:

* Separating composite attributes improves schema readability and ensures a flexible design that can accommodate future changes or enhancements. Separating the 'full\_name' attribute, for example, enables more efficient searches, such as finding patients or doctors with a specific last name.
* Removing derived attributes simplifies database design while improving performance by preventing inconsistencies and lowering storage requirements. For example, dynamically calculating the 'age' attribute using 'birth\_date' ensures accurate and up-to-date age information without the need to explicitly store it.
* Creating new entities for multivalued attributes keeps data requirements and relationships more accurate while providing a detailed and organized database structure. Creating the 'Patient\_Allergies' entity, for example, allows for easier tracking of multiple allergies per patient and simplifies allergy-related queries.

1. **Alternative Solutions and their Shortcomings:**

Because of their limitations and negative impact on database performance and organization, alternative solutions were not chosen. Alternative solutions and their drawbacks include:

* Storing composite attributes as a single attribute would restrict searching and organizing capabilities within the database, resulting in decreased performance and flexibility. For example, if 'full\_name' was stored as a single attribute, searching for all doctors with the surname "Smith" would be more difficult.
* Storing derived attributes directly in the schema introduces redundancy and the possibility of inconsistency, compromising data integrity and performance. For example, storing 'age' as an attribute may result in inaccurate information if the stored age is not updated promptly when a patient's 'birth\_date' changes.
* Storing multivalued attributes as comma-separated values within a single attribute, or creating multiple columns for the same attribute in the same entity, would slow query performance and data management, making it a less optimal choice than creating new entities. For example, if the Patient entity's 'allergies' attribute contained comma-separated values, querying patients with a specific allergy would be more complex and inefficient, and adding, removing, or modifying allergy information would be difficult and error-prone. If the 'allergies' attribute in the Patient entity had multiple columns, such as 'allergy\_1', 'allergy\_2', and so on, it could introduce a large amount of unused, reserved storage (null values), as well as new problems if the patient had more than two allergies.

Finally, the methods chosen for dealing with composite, derived, and multivalued attributes in the hospital database were founded on the principles of efficient database design and accurate data representation. The resulting schema is organized, flexible, and performance-optimized as a result of implementing these solutions, providing a solid foundation for meeting the hospital's data requirements and future enhancements.

Using examples from the original database and hypothetical scenarios, the following section discusses the reasons for identifying the types of relationships, how each relationship was handled, and the rationale behind these decisions.

1. **Types of Relationships:**

To ensure an accurate and efficient representation of the data requirements, it is critical to determine the relationships between entities and their cardinality ratios during the database design process. We classified the relationships as one-to-one, one-to-many, or many-to-many based on real-world scenarios and hospital data requirements. The following is a summary of how we decided on each type of relationship and the reasoning behind our choices:

* **One-to-One:** When one instance of one entity is associated with one instance of another entity, a one-to-one relationship exists. By analyzing the data requirements and recognizing that the entities involved have a unique and exclusive relationship, we were able to identify one-to-one relationships. Consider the relationship between an Operating\_Room and an OR\_Bed. Each operating room has a bed, and each OR\_Bed is only used in one operating room. We determined the one-to-one relationship by observing the data requirements and the unique nature of this relationship.
* **One-to-Many:** When one instance of one entity is associated with multiple instances of another entity, a one-to-many relationship exists. By analyzing the data requirements and recognizing that the entities involved have an association in which one entity instance can be related to multiple instances of the other entity, we identified one-to-many relationships. Consider the relationship between a Doctor and Surgery, for example. Each doctor may perform multiple surgeries, but only one doctor performs each surgery. We determined this to be a one-to-many relationship based on the hospital's data requirements, which include tracking surgeries performed by each doctor.
* **Many-to-Many:** When multiple instances of one entity can be associated with multiple instances of another entity, a many-to-many relationship exists. We discovered many-to-many relationships by analyzing the data requirements and recognizing that the entities involved have a more complex relationship in which multiple entity instances can be related to multiple instances of the other entity. Consider the relationship between a doctor and a patient. A doctor may have several patients, and a patient may be treated by several doctors. The hospital's data requirements include tracking the doctors in charge of each patient's treatment and vice versa. We identified this as a many-to-many relationship based on our understanding.

1. **Handling of Relationships:**

To create an organized and efficient schema, each relationship type was handled based on its cardinality:

* **One-to-Many:** These relationships were represented by inserting a foreign key into the "many" side of the relationship, which referred to the primary key of the "one" side. For example, consider the doctor-surgery relationship, in which one doctor can perform many surgeries, but each surgery is performed by a single doctor. The Surgery entity's foreign key 'doctor\_id' refers to the Doctor entity's primary key 'doctor\_id'.
* **Many-to-Many:** To address these relationships, an associative entity was created to split the many-to-many relationship into two one-to-many relationships. For example, consider the doctor-patient relationship, in which a doctor may have multiple patients and a patient may have multiple doctors. To represent this relationship, we created the 'Doctor\_Patient' associative entity with the foreign keys 'doctor\_id' and 'patient\_id'.
* **One-to-One:** These relationships were represented by including a foreign key in one of the entities that referenced the primary key of the other entity. For instance, consider the relationship between Operating\_Room and OR\_Bed, in which each operating room has one bed and each bed belongs to a single operating room. The OR\_Bed entity's foreign key 'operating\_room\_id' refers to the Operating\_Room entity's primary key 'operating\_room\_id'.

1. **Best Solutions Rationale:**

The following reasons led to the selection of the chosen methods for handling relationships as the best solutions:

* Handling one-to-many relationships with foreign keys provides a simple and efficient representation of real-world associations, allowing for easy querying and data management.
* Using associative entities to represent many-to-many relationships ensures a more accurate representation of data requirements and relationships, as well as a normalized database structure and efficient querying.
* Representing one-to-one relationships with foreign keys facilitates relationship management and ensures data consistency.

1. **Alternative Solutions and their Shortcomings**

Because of their limitations and negative impact on database performance and organization, alternative solutions to handling relationships in the database schema were not chosen. Here are some examples of how relationships were handled differently and their flaws:

* **One-to-Many:** Transferring the primary key from the "many" side to the "one" side. For example, in the Doctor-Surgery relationship, instead of a foreign key 'doctor\_id' in the Surgery entity, we could have a list of surgery IDs in the Doctor entity. The disadvantages of this approach include difficulty querying the data, inefficient storage, and potential inconsistencies, as it adds a multivalued attribute of 'surger\_id' to the Doctor entity, making the database more complex than necessary.
* **Many-to-Many:** There is no need to create a new associative entity. Instead of creating the 'Doctor\_Patient' associative entity in the relationship between Doctor and Patient, we could store a list of patient IDs in the Doctor entity and a list of doctor IDs in the Patient entity. Searching for all doctors associated with a specific patient or vice versa would require scanning and parsing the list of IDs in each entity, making querying complex and inefficient, along with a vast amount of redundant and duplicated data within the database. Maintaining consistency and updating relationships would also be difficult and error-prone.
* **One-to-One:** Keeping redundant data in both entities. Instead of having a foreign key 'operating\_room\_id' in the OR\_Bed entity, we could include information about the operating room in both the Operating\_Room and OR\_Bed entities in the relationship between Operating\_Room and OR\_Bed. Because the same information is stored in multiple locations, this approach introduces redundancy and potential inconsistencies. Changes to the information in the operating room would have to be propagated to both entities, increasing the risk of data inconsistency.

Finally, the methods chosen for dealing with relationships in the hospital database were founded on the principles of efficient database design and accurate data representation. The resulting schema is organized, flexible, and performance-optimized as a result of implementing these solutions, providing a solid foundation for meeting the hospital's data requirements and future enhancements.

* **Normalization:**

Normalization improves the effectiveness of a database by reducing data redundancy, ensuring data consistency, and simplifying the querying process. To achieve these objectives, we used normalization to the third normal form (3NF) in our database design. With examples from the schema, we will discuss how normalization improves the effectiveness of the hospital database.

1. **Reducing Data Redundancy:**

By dividing large tables into smaller, more manageable tables, normalization helps to reduce data redundancy. This separation prevents duplicate data from being stored and ensures that each attribute only contains atomic values.

In our hospital database, for example, we addressed the problem of multivalued attributes by creating a separate table for Patient\_Allergies (patient\_id, allergy) rather than storing the allergies in the Patient table. This decision ensures that the Patient table only contains atomic values, preventing data redundancy and inefficiency in storage.

1. **Ensuring Data Consistency and Integrity:**

A database's consistency and integrity are critical for its effectiveness. Normalization helps to keep data consistent by ensuring that attributes are dependent on their primary keys and removing transitive dependencies between non-prime attributes.

We discovered a transitive dependency between the insurance\_address attribute and the patient\_id attribute via the insurance\_company attribute in the hypothetical Patient\_Info relation. We ensured that non-prime attributes are no longer transitively dependent on their primary keys by splitting the Patient\_Info relation into two relations (Patient and Insurance). This method improves data consistency and integrity.

1. **Reducing Update Anomalies:**

Update anomalies, such as insertion, deletion, and modification anomalies, can result in database inconsistencies and inaccuracies. Normalization aids in the reduction of these anomalies by enforcing clear relationships and dependencies between attributes.

In the hypothetical Doctor\_Schedule relation, for example, we discovered a partial functional dependency in which certain attributes were only dependent on a portion of the primary key. We eliminated this partial dependency by splitting the relation into two (Doctor and Schedule), reducing the likelihood of update anomalies and ensuring that data remains accurate and consistent.

1. **Simplifying Querying Process:**

Normalization facilitates querying by organizing data in a structured manner, allowing for efficient information retrieval. Users can quickly and easily access and analyze data with smaller, well-structured tables.

We have created separate tables in our hospital database for entities such as Doctor, Patient, Operating\_Room, and Surgery to ensure that information is organized in a structured manner. This separation allows users to more easily retrieve and analyze specific information, contributing to the database's overall effectiveness.

To summarize, normalization is critical for improving the effectiveness of the hospital database by reducing data redundancy, ensuring data consistency and integrity, minimizing update anomalies, and simplifying the querying process. We created a robust and efficient schema that accurately represents the hospital's data needs and allows for simplified querying and data management by using normalization to the third normal form (3NF) in our database design.

* **Physical Design:**

The physical design of a database is critical to its effectiveness, performance, and usability. We ensure that the data stored in the database is accurate, consistent, and efficiently managed by selecting appropriate data types and constraints for each attribute. Let us talk about how the physical design of our hospital database schema, as well as the data types and constraints we use for each attribute, contribute to its overall effectiveness.

* 1. **Patient:**
* patient\_id: Making the primary key an auto-incrementing integer ensures uniqueness and simplifies data entry for new patients.
* first\_name, last\_name: The NOT NULL constraint ensures that each patient's first and last name are stored in the database, allowing for complete and accurate patient identification. The use of varchar(50) allows for the storage of names of varying lengths.
* middle\_name: Allowing NULL values for middle\_name allows patients who do not have a middle name to be accommodated. The use of varchar(50) allows for the storage of names of varying lengths.
* birth\_date: The NOT NULL constraint ensures that each patient has a birth date stored in the database, allowing for the calculation of the patient's age, which is critical for their treatment. The date data type stores patient birth dates accurately and efficiently.
* gender: Using the NOT NULL constraint to limit values to 'M' and 'F' ensures that all patients have a gender assigned, maintaining data consistency and preventing inaccuracies.
* phone\_number: The char(10) data type, combined with the NOT NULL constraint, ensures that each patient has a valid phone number.
* address: The use of varchar(150) allows for the storage of various address lengths, accommodating different patient address formats.
  1. **Doctor:**
* doctor\_id: Using an auto-incrementing integer primary key, similar to patient\_id, ensures uniqueness and simplifies data entry for new doctors.
* first\_name, last\_name: The NOT NULL constraint ensures that each patient's first and last name are stored in the database, allowing for complete and accurate patient identification. The use of varchar(50) allows for the storage of names of varying lengths.
* middle\_name: Allowing NULL values for middle\_name allows doctors who do not have a middle name to be accommodated.
* specialty: varchar(100) allows for a wide range of specialties while also providing flexibility in storing specialty names.
* phone\_number: As with patients, using char(10) and enforcing the NOT NULL and unique constraints ensures that each doctor has a unique and valid phone number.
  1. **Operating\_Room:**
* operating\_room\_id: For new operating rooms, an auto-incrementing integer primary key ensures uniqueness and simplifies data entry.
* room\_number: The use of char(4) and the NOT NULL constraint for room\_number creates a standardized format for storing room numbers and ensures that they are always provided.
* room\_type: varchar(50) allows for the storage of multiple room types.
* capacity: The int data type, which has a default value of 5, ensures that capacity values are integers and allows for simple room capacity adjustments.
  1. **Surgery:**
* surgery\_id: An auto-incrementing integer primary key ensures uniqueness and simplifies new surgery data entry.
* surgery\_type: varchar(100) allows for a variety of surgery types as well as flexibility in storing surgery names.
* start\_time, end\_time: The datetime data type stores surgery start and end times accurately, allowing for efficient scheduling and time calculations. The NOT NULL constraint is important for the start time but not for the end time because the start time is critical for OR schedules and the end time is unknown when the surgery is first scheduled, so it can contain a NULL value.
* doctor\_id, patient\_id, operating\_room\_id: Using foreign keys connects surgeries, doctors, patients, and operating rooms, ensuring data consistency and accurate referencing.
  1. **OR\_Bed:**
* bed\_size: Limiting values to 'S', 'M', and 'L' and char(1) ensures bed size standardization, simplifying inventory management.
* maximum\_weight: Using the int data type and setting a default value of 150 ensures a consistent format for maximum weight values.
* operating\_room\_id: Using a primary and foreign key ensures that beds and operating rooms are correctly referenced and related.
  1. **Patient\_Allergies:**
* patient\_id: This attribute ensures that each allergy is correctly associated with a specific patient, allowing for more accurate mapping of patients and allergies.
* allergy: The varchar(100) data type allows for the storage of various types of allergies, enabling comprehensive allergy tracking and management in the hospital database.
  1. **OR\_Equipment:**
* operating\_room\_id: This attribute ensures that each piece of equipment is correctly assigned to an operating room, allowing for better inventory tracking and management.
* equipment: The varchar(100) data type allows for the storage of various types of equipment, enabling comprehensive equipment tracking and management in the hospital database.
  1. **Doctor\_Patient:**
* patient\_id, doctor\_id: Using primary and foreign keys for both attributes ensures accurate relationships between doctors and patients, as well as data consistency and duplication prevention.
* appointment\_date: The date data type stores appointment dates accurately, allowing for efficient querying and appointment scheduling.
* appointment\_time: The time data type accurately stores appointment times, allowing for proper hospital scheduling and time management.

Overall, the physical design of this hospital database schema is effective due to the appropriate use of data types and constraints for each attribute. The design maintains data consistency, accuracy, and efficiency by enforcing relationships between entities and ensuring proper data validation. This results in a robust database that enhances the overall performance and usability of the hospital management system.

* **User and System Requirements:**

It is critical to consider various factors that contribute to the overall success of the database when evaluating the effectiveness of the database design in relation to user and system requirements. The factors can be divided into three categories: functionality, performance, and maintainability.

1. **Functionality:**

* **Completeness:** The designed database should include all necessary entities, attributes, and relationships to represent the data requirements of the organization. Patients, doctors, operating rooms, surgeries, OR beds, patient allergies, doctor-patient appointments, and OR equipment are all part of the hospital database design. This thoroughness ensures that the database addresses the most important aspects of hospital operations.
* **Consistency:** The use of primary and foreign keys, as well as defined relationships between tables, aids in the maintenance of data consistency throughout the database.
* **Integrity:** To maintain data integrity, prevent incorrect data entry, and ensure data consistency, the database design enforces constraints such as NOT NULL, UNIQUE, DEFAULT, AUTO\_INCREMENT, and VALUES.

1. **Performance:**

* **Query Efficiency:** By utilizing appropriate data types, indexing strategies, and relationship structures, the database design promotes efficient querying. The use of int for IDs and date/time data types for date and time attributes, for example, allows for faster querying and value comparison.
* **Storage Efficiency:** By allocating space only for the actual length of stored strings, the use of appropriate data types such as varchar and char ensures efficient storage space utilization.
* **Scalability:** The database design allows for potential growth in the number of records, as the auto-increment feature in primary keys accommodates new entries without requiring manual intervention.

1. **Maintainability:**

* **Modularity:** The database design is organized into distinct tables that represent entities, making it easier to understand and manage. This modularity simplifies future updates and modifications.
* **Flexibility:** The design accommodates changes in the organization's data requirements by allowing for the addition of new entities, attributes, and relationships, as well as the modification of existing ones.
* **Documentation:** The three stages of the design process (conceptual, logical, and physical) have been clearly outlined and documented, providing a solid foundation for future modifications and troubleshooting.

Finally, the hospital database design appears to be effective in meeting the needs of users and systems by providing functionality, performance, and maintainability. While there is always room for improvement, this design provides a solid foundation for a hospital database system that can evolve to meet changing needs and requirements over time.

**Conclusion:**

Finally, this report provided a thorough examination of the hospital database design process, beginning with an overview of the conceptual, logical, and physical stages of the design process. The hospital database accurately records key aspects of hospital operations such as patients, doctors, operating rooms, surgeries, and more. The database's design prioritizes data consistency, integrity, performance, and maintainability, ensuring that it can meet the diverse needs of users and the system itself.

The database design's success in meeting user and system requirements demonstrates its potential to serve as a solid foundation for hospital management and decision-making processes. The design, with its modular, flexible, and well-documented structure, allows for future expansion and adaptation as organizational needs change. While continuous monitoring and improvement are essential for optimal performance, the presented design serves as a solid foundation for a hospital database system capable of supporting efficient operations and growth.

It is recommended that the hospital database be assessed and optimized on a regular basis in order to keep up with technological advancements, changing user requirements, and industry best practices. As a result, the hospital can maintain a dependable, high-performing database system that supports its core operations, improves patient care, and promotes informed decision-making throughout the organization.

# ***References***

Abba, I.V. (2022) *Crow’s Foot Notation – Relationship Symbols And How to Read Diagrams, Free Code Camp.* Available at: <https://www.freecodecamp.org/news/crows-foot-notation-relationship-symbols-and-how-to-read-diagrams/>

Banister, R. (2021) *Physical Database Design | Sesame Software & Relational Junction, Sesame Software.* Available at: <https://sesamesoftware.com/white-papers/physical-database-design/>

Borgida, A., Casanova, M.A. and Laender, A.H.F. (2016*) Logical Database Design: From Conceptual to Logical Schema, Encyclopedia of Database Systems.* Available at: <https://doi.org/10.1007/978-1-4899-7993-3_645-2.>

Broadcom (2013) Introduction to Logical Design, Broadcom. Available at: <https://techdocs.broadcom.com/us/en/ca-mainframe-software/database-management/ca-idms/19-0/administrating/database-design/introduction-to-logical-design.html>

Chola, A. (2022) *Overview of a Physical Design Database: 5 Critical Aspects, Hevodata.* Available at: <https://hevodata.com/learn/physical-design-database/>

Chris, K. (2021) *Database Normalization – Normal Forms 1nf 2nf 3nf Table Examples, Free Code Camp.* Available at: [https://www.freecodecamp.org/news/database-normalization-1nf-2nf-3nf-table-examples/#whatisdatabasenormalization](https://www.freecodecamp.org/news/database-normalization-1nf-2nf-3nf-table-examples/%23whatisdatabasenormalization)

Dybka, P. (2016) *Crow’s Foot Notation, Vertabelo Database Modeler.* Available at: <https://vertabelo.com/blog/crow-s-foot-notation/>

GeeksforGeeks (2011) *Logical Database, SpringerReference.* Available at: <https://doi.org/10.1007/springerreference_17948.>

IBM (2017) *Physical database design, IBM.* Available at: <https://www.ibm.com/docs/en/db2-for-zos/11?topic=relationships-physical-database-design>

IBM (2019) *What is a database schema?, IBM.* Available at: [https://www.ibm.com/topics/database-schema](https://www.ibm.com/topics/database-schema%20)

Informatica (2019) *What is Data Mapping?, Informatica.* Available at: [https://www.informatica.com/resources/articles/data-mapping.html](https://www.informatica.com/resources/articles/data-mapping.html%20)

Informatica e Ingegneria Online (2020) *Difference between conceptual, logical and physical design of a database, Informatica e Ingegneria Online.* Available at: [https://vitolavecchia.altervista.org/difference-between-conceptual-logical-and-physical-design-of-a-database/](https://vitolavecchia.altervista.org/difference-between-conceptual-logical-and-physical-design-of-a-database/%20)

Javatpoint (2021) *DBMS Normalization: 1NF, 2NF, 3NF and BCNF with Examples, Javatpoint.* Available at: <https://www.javatpoint.com/dbms-normalization>

Keating, G. (2021) *Data Mapping 101: What It Means and How to Do It, Segment.* Available at: <https://segment.com/blog/data-mapping/>

Lucidchart (2017) *What is a Database Schema, Lucidchart.* Available at: <https://www.lucidchart.com/pages/database-diagram/database-schema>

MariaDB (2016) *Database Design Phase 2: Conceptual Design, MariaDB.* Available at: [https://mariadb.com/kb/en/library/database-design-phase-2-conceptual-design/](https://mariadb.com/kb/en/library/database-design-phase-2-conceptual-design/%20)

ORACLE (2022) *MySQL 8.0 Reference Manual - 11 Data Types, MySQL 8.0 Reference Manual.* Available at: [https://dev.mysql.com/doc/refman/8.0/en/data-types.html](https://dev.mysql.com/doc/refman/8.0/en/data-types.html%20)

Peterson, R. (2022) *What is Normalization in DBMS (SQL)? 1NF, 2NF, 3NF Example, Guru99.* Available at: [https://www.guru99.com/database-normalization.html](https://www.guru99.com/database-normalization.html%20)

Qiu, M. et al. (2022) *Database normalization description - Office | Microsoft Learn, Article.* Available at: [https://learn.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description](https://learn.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description%20)

StudyTonight (2020) *1NF, 2NF, 3NF and BCNF in Database Normalization | Studytonight, StudyTonight.* Available at: [https://www.studytonight.com/dbms/database-normalization.php](https://www.studytonight.com/dbms/database-normalization.php%20)

Talend (2022) *What is Data Mapping? Definition and Examples, Talend.* Available at: [https://www.talend.com/resources/data-mapping/](https://www.talend.com/resources/data-mapping/%20)

Thiru (2016) *Conceptual Design in Database Design Process, My Reading Room.* Available at: [http://www.myreadingroom.co.in/notes-and-studymaterial/65-dbms/507-conceptual-design.html](http://www.myreadingroom.co.in/notes-and-studymaterial/65-dbms/507-conceptual-design.html%20)

Watt, A. (2014) *Chapter 8 The Entity Relationship Data Model – Database Design – 2nd Edition, opentextbc.* Available at: <https://opentextbc.ca/dbdesign01/chapter/chapter-8-entity-relationship-model/>