2024

CMSC424: Database Design

MIDTERM REVIEW GUIDE

RISHABH BARAL

Table of Contents

Topic 1: (Extended) Entity Relationship Diagrams

Subtopic 1.1: Entity Relationship Diagrams: Defining Terms

The **Entity Relationship model** describes data as a combination of three things: Entities, Attributes, and Relationships. An **entity set** is a collection of entities that all share the same properties (**attributes**) and are key to the organization of the data in the dataset. An **entity** is a **SINGLE OCCURRENCE** of an entity set.

An entity set has a relationship with an entity in an analogous manner to object-class relations in Object-Oriented Programming. Essentially, **an entity is to an entity set** what **an object is to a class.**

Key Vocabulary

Term	Definition
Entity Set	Collection of Entities that all share the same properties
Entity	An occurrence of an entity set
Attribute	A property or characteristic of an entity or entity set

Subtopic 1.2: Entity Relationship Diagrams: Introducing the Diagram

In an Entity-Relationship Diagram, **an entity set is represented by a rectangle**. Continuing with the analogy of Entity Sets being classes and Entities being objects, if we recall from the many previous programming courses we may have taken (CMSC131, CMSC132, CMSC216, CMSC330, etc.), every class has a set of variables. Just as classes have variables, so do entity sets. The "variables" that describe entities are called **attributes.** In the Entity-Relationship Diagram, the attributes of an entity set are listed **inside the rectangle for the entity with which the attribute(s) are associated.**

These attributes themselves can satisfy many different conditions. An attribute can be either Simple or Composite. **Simple Attributes** are those that can **NOT** be divided any further. For this reason, simple attributes are also called "**atomic attributes**". **Composite Attributes,** on the other hand, are those that are made up of **multiple constituent fields** and are often **divided for easier storage.** In an Entity-Relationship Diagram, Composite Attributes are shown by either **listing the attribute with its constituent parts indented** or with the **attribute name followed by its constituent parts in parentheses.**

Attributes can also be either single-valued or multi-valued. A **single-valued attribute** as the name suggests is an attribute that only takes on one value. A **multi-valued attribute,** analogously, is an attribute that can have multiple values. In the Entity-Relationship Diagram, multi-valued attributes are represented with either the attribute name in **square brackets** or **curly braces.**

Further, to save space and reduce redundancy, not all attributes are stored in a dataset. Some attributes can be derived from others already present. Such attributes, as the description

suggests, are called **derived attributes**. On an Entity Relationship Diagram, derived attributes are presented with the attribute name **followed by** a set of **EMPTY** parentheses.

Entity Sets and the attributes they contain are often designed to reflect what users would enter into a form. As such, attributes can be **optional** or **required**. An **optional attribute**, as the name suggests, is one that is **NOT** required to have a value. In an entity relationship diagram, required attributes are indicated in boldface text with optional attributes indicated in normal font.

Since Entity Sets can be thought of as "classes" the entities associated with a particular entity set must be **unique**. Typically, one or more attributes in an entity will be uniquely able to identify the entity. These attributes are called the entity's **identifier** or **key.** In an Entity-Relationship Diagram, the **identifier** or **key** for an entity set will be **underlined**. Furthermore, just as attributes can be made of more than one value, so can keys. A **composite key** is a key or identifier for an entity set that requires the use of more than one attribute to uniquely identify the entities of the set.

Attributes are fields that are stored for each entity of an entity set. As such, cases may arise where placing a limit on the values that an attribute can take may be beneficial. The **value set** of an attribute is the domain (range of values) that are allowable for an attribute.

In addition, Entity Sets themselves may be dependent on other Entity Sets for relevancy (ex. An employee has dependents). In such a case, the standalone entity set is called a **strong entity set** while the dependent entity set is called a **weak entity set**. In an Entity Relationship Diagram, strong entity sets are indicated by rectangles with a single outer border whilst weak entity sets are indicated by rectangles with a **double outer border.**

Key Vocabulary

Term	Definition
Simple Attribute	An Attribute that cannot be broken any further a.k.a "atomic attributes"
Composite Attribute	An Attribute that is made of different parts, and is often broken into its constituent elements
Single-Valued Attribute	An attribute that can take on a single value
Multi-Valued Attribute	An attribute that can take on more than one value
Derived Attribute	Any attribute or piece of information that can be **derived** from the information already present in the entity set
Key	An attribute or set of attributes that can be used to **uniquely identify** instances of an entity set
Composite Key	A Key involving multiple attributes
Value Set	The range of values allowable for an attribute of an entity set
Strong Entity Set	An Entity set that can **stand alone** without depending on any others
Weak Entity Set	An entity set that cannot exist without the presence of a Strong Entity Set, on which it is Dependent

Subtopic 1.3: Entity Relationship Diagrams: Understanding Relationships

A **relationship** is a meaningful association between one or more entities. A **relationship set** is a group of relationships of the same type (between the same entity sets). In the Entity-Relationship Diagram, relationships are represented as **diamonds** with lines connecting the diamond to **each** entity set involved in the relationship. Regardless of the *verb* in the diamond for a relationship, the relationship can be read in both directions, being read from **right to left** or from **left to right.** The **degree** of a relationship is determined by the number of entity sets that make up the relationship. If one entity set is involved, the relationship is **unary,** if two entity sets are involved, the relationship is **binary,** and the relationship is **ternary** if three entity sets are involved.

In addition, the **cardinality ratio** or **mapping cardinality** of a relationship specifies how many instances of one entity set can be connected to how many instances of another entity set via a relationship. Since most relationships are binary, the four most common cardinality ratios are: **"one to one", "one to many", "many to one",** and **"many to many"**. As specified in the textbook and in Herve's Slides, the notation used will be that a vertical bar (**|**) indicates "one" while a crow's foot (**<**) indicates "many". Furthermore, a relationship may itself have attributes which are associated with **none** of the entity sets involved. An attribute of a relationship is indicated as a separate rectangle attached to the relationship diamond by a **dotted line**. Just as relationships can have cardinality ratios, they can also have cardinality constraints. Any entity can be "mandatory" or "optional" with respect to the relationship. In the Entity-Relationship Diagram, if an entity is to be optional in a relationship, it is indicated with an oval (**0**) **immediately before or after the cardinality ratio**. . In the Entity-Relationship Diagram, if an entity is to be mandatory in a relationship, it is indicated with a vertical bar (**|**) **immediately before or after the cardinality ratio**.

If a relationship when represented in an Entity-Relationship diagram has **multiple** attributes of the relationship, then it is best to create a separate entity and represent the relationship as what is known as an **associative entity set**.

Key Vocabulary

Term	Definition
Relationship	A meaningful association between **one or more** entities
Relationship Set	A group of relationships of the same type (i.e., Involving the same entity sets)
Degree	The number of entity sets involved in a relationship
Unary Relationship	A Relationship involving members of **one** entity set
Binary Relationship	A Relationship involving members of **two** entity sets
Ternary Relationship	A Relationship involving members of **three** entity sets
Optional One	one entity instance **may or may not be** associated with **exactly one** instance of another entity
Optional Many	one entity instance **may or may not be** associated with **multiple** instances of another entity
Mandatory One	one entity instance **must** be associated with **exactly one** instance of another entity
Mandatory Many	one entity instance **must** be associated with **one or more** instances of another entity.

Associative Entity Set	A separate entity set created to represent relationships with multiple attributes

Subtopic 1.4: Entity Relationship Diagrams: Extended Entity Relationship

Even though entities from the same entity set share attributes, it is not uncommon to want to split the entity set to keep track of unique groups of individuals, each with their own unique set of attributes. This is done using what are called **subsets** or **subtypes** in a diagram known as an Extended Entity-Relationship Diagram. Within these subsets, they may overlap (A member of one entity can be a member of another entity simultaneously). This is denoted in an EER diagram with a tiny **o** placed at the juncture of the entities in question. Should the members of these entities be disjoint (members of one entity can **NOT** be a part of a second separate entity), the analogous representation is to place a tiny **d** at the juncture point between the entities in question. In addition, the combination of constituent entity subsets may not entirely generate the entity set from which the subsets are derived. In such a case, this is known as **partial specialization**. In the event that the chosen subsets of the entity set completely cover the entity set from which the subsets are derived, the decomposition is said to satisfy the property of **completeness** or **complete specialization**. On an Extended Entity-Relationship Diagram, completeness is indicated by a **double-weighted** line while partial specialization is indicated with a **single-weighted** line. Due to the subset-superset nature of the Extended Entity Relationship model, the hierarchical structure of an EER Diagram resembles a **tree**.

Key Vocabulary

Term	Definition
Subset (Subtype)	Entity set(s) created by splitting a larger entity set to group entities with similar attributes
Overlap	A member of one entity (subset/subtype) can be a member of a second entity simultaneously
Disjointness	A member of one entity (subset/subtype) **cannot** be a member of a second entity simultaneously
Partial Specialization	Combining Subsets **may not** generate the entity set from which the subsets were originally generated
Completeness	Combining Subsets **will** generate the entity set from which the subsets were originally generated

Topic 2: Logical Design and Functional Dependencies

Subtopic 2.1: Designing a Database

In the Relational Database model, we represent data as tables. This idea has mathematical foundations and is often understood as the combination of three concepts: Data Structure (rows and columns), Data Manipulation (SQL), and Data Integrity (Rules to enforce constraints).

A **relation** is a **2-D table** consisting of a set of **named** columns and an **arbitrary** number of rows. Every table in a database has a unique name and values stored in the table's cells (row-column intersections) must be **atomic**.

Just as entities had something unique to identify them, so too do relations. Each attribute of a relation must be unique **WITHIN THAT RELATION ONLY.** The sequence in which columns are presented (left to right) is **NOT SIGNIFICANT.** Analogously, the sequence in which rows are presented (top to bottom) is **NOT SIGNIFICANT.** A **table** can have any number of relations. Relations are represented by their name, followed by a **comma-delimited** set of attributes within the relation.

In a relation, the **primary key** or **identifier** is indicated by an underline. The primary key or identifier may be a single attribute but can also be a collection of attributes. The primary key itself has a few constraints. For example, the primary key of a relation must contain **unique** values, but only with regard to the relation for which it is the primary key. Furthermore, the column(s) identified to be used as the primary key **CANNOT** contain a **NULL** value. In theory, every relation (table) should be created with a primary key. However, in practice, languages like MySQL allow for relations to be created without a specified key. It is desirable to have a primary key to make references easier, so in practice, at least in the context of CMSC424, we will always create relations **WITH** a primary key.

When designing relations, we want them to be structured. However, this isn't always easy. Many relations are inadvertently designed with **redundancy**, where data in the table (Relation) is easily noticeable as **duplicated** and **repetitive**. Other relations are designed with an **insertion anomaly** whereby inserting a new element would result in a violation of a key constraint (most often the **primary key constraint**). Another common type of issue with creating relations is what is known as a **deletion anomaly** whereby deleting an element from the relation would result in the loss of information associated with that element that appears **nowhere else** in the table. Lastly, relations may also be designed with an inadvertent **update anomaly** whereby updating the values associated with one element may require updates in **several rows** containing data from the same element (based on primary key). The solution is to split the relation into various tables, maintain references to each other via **foreign keys**.

To maintain structure in relations, there are a few common constraints that are abided by in practice. Firstly, **domain constraints** constrain the values placed in the table by ensuring that the values are all **a certain data type, do not exceed a maximum length, etc.** The values follow the domain constraints if they are from the **same domain**. Secondly, **entity integrity constraints** constrain the tables themselves by ensuring that **every primary key attribute is non-null (both for**

atomic and composite primary keys) and by ensuring that **every row in the table has a different UNIQUE value for the primary key attribute(s).** Lastly, **referential integrity constraints** ensure that references between tables are made correctly. To adhere to referential integrity constraints, association between any **two or more** separate tables must be done through a **foreign key**, which is an attribute or set of attributes in one table that references the primary key of another table. Furthermore, every value of a table's foreign key must either be **null** or **be present as a value of the primary key** in the table which is referenced using the foreign key.

Key Vocabulary

Term	Definition
Relation	A 2-D table consisting of named columns and an arbitrary number of rows
Redundancy	Data in a table (Relation) is duplicated and repetitive
Insertion Anomaly	Situation in which insertion of an element violates Primary Key Constraint
Deletion Anomaly	Situation in which deletion of an element results in loss of information
Update Anomaly	Situation in which updating an element requires updating several rows
Domain Constraints	Constraints specifying the format of values for the table (Data type, maximum field length, etc.)
Entity Integrity Constraints	Constraints specifying the design of tables (Primary key cannot be Null **AND** primary key values must be unique)
Referential Integrity Constraints	Constraints enforced to ensure references between tables are correct (Association between tables must be by foreign key, Foreign key value must be **null** or be **present as primary key value** in referenced table)
Foreign Key	an attribute or set of attributes in one table that references the primary key of another table.

Subtopic 2.2: Functional Dependencies

A **functional dependency** in simplest terms can be described as a relation between attributes or fields in a record. Not every functional dependency is unique. Having a functional dependency between attributes of a record just means that whenever one of the linked values is searched, the other will automatically pop up. Functional dependencies between two attributes A and B are written with an arrow between the attributes, appearing as follows: **a→b**. The attribute on the left hand side of a functional dependency is called the **determinant** and the attribute on the right hand side of a functional dependency is called the **dependent**.

To understand and track functional dependencies accurately, we will abide by three rules known as **Armstrong's axioms** and three further rules known as the **derived axioms**. The first of Armstrong's axioms is the most straightforward. Called the **reflexivity axiom**, it states that **if there are two sets of attributes A and B such that B is contained within A, then A implies B**. The second Armstrong axiom is called the **axiom of augmentation**, and it can be encapsulated simply by the phrase **"It isn't really necessary to have this here, but it doesn't hurt"**. In a more formal definition of the augmentation axiom, we can say that **if there are three sets of attributes A, B, and C with the known relation that A implies B, then we can safely say that AC implies BC since A implies B already holds**. The third and final Armstrong Axiom is called the **axiom of**

transitivity and derives from the transitive property often used in algebraic set theory. In essence, **if there exist three sets of attributes A, B, and C such that A implies B and B implies C, then the transitivity axiom can be used to say that A implies C**. From the three Armstrong Axioms, we can derive three further rules that will help in understanding functional dependencies as they exist between attributes and entity sets. Firstly, we can derive the **union rule,** which states that **if a relation contains the functional dependencies A implies B** AND **A implies C, then this can simply be restated as A implies both B and C**. Secondly, we can derive the **decomposition rule** which states **that if a relation contains the functional dependency A implies BC, then this can be simply restated as two dependencies: A implies B** AND **A implies C**. Lastly, we can derive the **pseudo-transitivity rule** which states that **if a relation contains the functional dependencies A implies B** AND **BC implies D, then this can simply be restated as AC implies D**.

Now that we understand what functional dependencies are and the rules that govern the relations between them, we can delve into the concept of **closure**. Closure is described as **the set of all functional dependencies logically implied by an attribute or set of attributes.** To determine the closure of an attribute or set of attributes, the simplest process is to first **add the attribute or set of attributes into a "result set".** The second step is to **recursively use the attribute or set of attributes in the "result set" along with any provided functional dependencies to determine which attribute(s) to add to the "result set".** After the completion of the second step, the "result set" should contain the full closure of the desired attribute or set of attributes. If the closure of an attribute or set of attributes contains all the attributes in the relation from which the closure was generated, then that attribute or set of attributes is called the **super key** of the relation. A super key becomes a **candidate key** if there is no subset of the chosen set of attributes that can generate the entire relation solely through its own closure.

Another concept that can directly be inferred from the understanding of functional dependencies and the rules that govern the relations between them is the idea of what is known as the **canonical cover** of a relation. The canonical cover of a relation is described as the **minimal set of functional dependencies required to obtain the entire relation through closure of those functional dependencies.** The algorithm for finding the canonical cover for a relation is a 4 step process. Firstly, decompose the list of functional dependencies provided in a way that ensures that each functional dependency has only **ONE** attribute on the **right hand side** of the chosen functional dependency. Secondly, eliminate from the set all trivial dependencies (those implied by the reflexivity axiom in Armstrong's Axioms). Thirdly, eliminate all dependencies that can be deduced by combining two or more dependencies in the set (typically these are the dependencies that would be implied by the Transitivity Axiom). Lastly, combine dependencies that have the same **attribute** on the **left hand side** of the dependency. In doing so, you are first expanding the set of dependencies to see all the dependencies present (including ones that may not have been explicitly stated to begin with) and then reducing the set of dependencies down to its canonical cover by eliminating redundant dependencies in the set.

Key Vocabulary

Term	Definition
Functional Dependency	A relation between two attributes or fields of a record
Determinant	The attribute(s) on the **left hand side** of a FD
Dependent	The attribute(s) on the **right hand side** of a FD
Closure	The set of all FDs logically obtainable from a chosen attribute or set of attributes
Super key	Attribute or set of attributes whose closure is the entire relation
Candidate Key	A Super Key of which no subset can generate the entire relation through ***only its closure***
Canonical Cover	The minimum set of Functional Dependencies required to generate the entire relation

Key Theorems/Algorithms

Concept	Formula/Algorithm
Reflexivity Axiom	If $B \in A$, then $A \rightarrow B$
Augmentation Axiom	If $A \rightarrow B$, then $AC \rightarrow BC$ **For any C such that $C \neq A$ and $C \neq B$**
Transitivity Axiom	If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$
Union Rule	If $A \rightarrow B$ and $A \rightarrow C$, then $A \rightarrow BC$
Decomposition Rule	If $A \rightarrow BC$, then $A \rightarrow B$ and $A \rightarrow C$
Pseudo-Transitivity Rule	If $A \rightarrow B$ and $BC \rightarrow D$, then $AC \rightarrow D$
Closure of an Attribute Set	1. Add the attribute(s) chosen to a result set 2. Recursively calculate the closure by using the attributes in the result set and the **6** rules.
Canonical Cover	1. Decompose the list of FDs such that each has a single attribute as the dependent. 2. Eliminate all trivial dependencies from the set 3. Eliminate all transitive dependencies 4. Combine all FDs with the same determinant

Topic 3: Entity Relationship Diagrams to Tables

Subtopic 3.1: Entities

When converting an Entity Relationship diagram to a **relational schema,** which is defined as a **set of tables that when combined show the entire relation**, the first thing that must be converted are the entities themselves. As you might recall from designing an ER diagram, entities can be **Strong (standalone)** or **weak (dependent on one or more entities)**. This results in two different processes for converting entities when creating relational schema.

Let's first discuss the process for converting Strong Entity Sets to relations since it is considerably easier to do so. In essence, the main principle to remember is that **each entity set becomes a table.** Most attributes for the entity should be converted to columns in the new table. Do **NOT** create columns for derived attributes, as these values are not intended to be stored. Do not create columns for multivalued attributes; we will address these later. For composite attributes, create columns only for the component attributes, not the composite itself. As with entities, you will need to decide on a name for each new column, which does not have to be the same as the attribute name. You will also need to specify a type and any constraints for the column. Choose a key attribute (every regular entity should have at least one) and use the column created from it as the primary key for the new table. If the entity has multiple key attributes, you will need to decide which one makes most sense as a primary key. Simpler primary keys are usually preferred over more complex ones.

Now that we know how to convert strong entity sets to relations, we can move to the more difficult task of converting weak entity sets to relations. We will do this in nearly the same way as regular entities. However, recall that a weak entity has no identifying key attribute. Instead, it has a partial key, which must be combined with the key of the parent entity. The table created from a weak entity must therefore incorporate the key from the parent entity as an additional column. The primary key for the new table will be composed of the columns created from the parent key and from the partial key. Additionally, the column created from the parent key should be constrained to always match some key in the parent table, using a foreign key constraint.

Subtopic 3.2: Relationships

Logically, it follows that once the entities themselves have been converted, the next thing to convert would be the relationships between those entities. There are three main kinds of relationships, but they can all be derived from a single type of relationship: **many to many**. To convert a many-to-many relationship, we must remember that many-to-many relationships are the most general type of relationship; a database structure accommodating a many-to-many relationship can also accommodate one-to-many or one-to-one relationships, as "one" is just a special case of "many". The challenge for many-to-many relationships is how to represent a connection from a record in one table to multiple records in the other table. While modern SQL allows array valued columns in tables, not all databases support them. The traditional solution is to create a cross-reference table. Given a table A and a table B, we create a cross-reference table with columns corresponding to the primary keys of A and B. Each row in the cross-reference table stores one unique pairing of a primary key value from A with a primary key value from B. Each row thus

represents a single connection between one row in A with one row in B. If a row in A is related to multiple rows in B, then there will be multiple entries with the same A primary key value, paired with each related B primary key value.

Just as many-to-many relationships are represented using a cross-reference table, so too can one-to-many relationships. However, if we realize that rows on the “many” side of the relationship can be associated with at most one row from the “one” side, we can choose to capture the relationship by storing the primary key of the “one” side table in the “many” side table.

Further, a one-to-one relationship is a special case of both a one-to-many relationship and a many-to-many relationship. For this reason, there is no one right way to store a one-to-one relationship in a relational schema. However, in most cases, it will be preferable to borrow the primary key from one table as a foreign key in the other table. Using this approach, you could borrow from either side; however, one choice is often preferable to another, depending on how the relation is represented in the ER diagram and on what makes the most logical sense.

Subtopic 3.3: Supertypes and Subtypes

Subtypes and Supertypes are an extension of the entity model used when creating a database. Often it is useful to split an entity into various subtypes to be able to accurately convey more information and retain a less redundant database. To do so, the most common way of storing subtypes, supertypes, and their relations in a relational schema is to create one relation for the supertype and then create relation(s) for the subtype(s), remembering to put the primary key of the supertype in the relation(s) to be used as a foreign key for the subtypes. Firstly, all relations representing entity types in the same hierarchy have the same primary key. Secondly, the primary key of a subtype relation will also be a foreign key that references its supertype relation. Lastly, attributes of a supertype (except for the primary key) appear only in the relation that represents the supertype.

Topic 4: Normalization

Subtopic 4.1: What is Normalization?

Before we delve into the specifics of normal forms, I would like to provide some preliminary insights into why we normalize data when we store it. Edgar F. Codd determined that relations stored without caring for how the data was stored caused immense problems when updating the data in those relations. Codd discovered that data is often stored redundantly which wastes space. As such, he suggested that the data be **normalized** to avoid these anomalies as much as possible. Now you might be wondering what those anomalies are and there are three common anomalies: Insert Anomalies, Update Anomalies, and Delete Anomalies. The **update anomaly** refers to any situation in which updating the value for **one attribute** in **one record** requires the update of **MULTIPLE** rows. The **insert anomaly** refers to the situation in which a new record cannot be inserted into the database due to an unforeseen dependency on another relation within the database. Lastly, the **deletion anomaly** refers to the situation in which a record cannot be deleted from a database without either losing information, related records, or both.

In this way, normalizing a database and its data helps the data to remain safe and usable to retrieve all necessary information with minimum redundancy. Now that we know the motivation behind normalizing data, we can discuss what normalization really is. **Normalization** is the process of organizing data in a database by creating tables and establishing relationships between those tables using rules that are designed to protect the data by eliminating redundancy and inconsistent dependency.

Key Vocabulary

Term	Definition
Update Anomaly	Any situation in which updating the value(s) of a record would require multiple updates
Insert Anomaly	Any situation in which a new record cannot be inserted due to an unforeseen dependency
Delete Anomaly	Any situation where a record cannot be deleted because deletion would result in a loss of data
Normalization	The process of organizing data into a database by creating tables and establishing relations between those tables

Subtopic 4.2: Normal Forms (1NF, 2NF, 3NF)

The basic normal form is called First Normal Form or 1NF. The first normal form follows logically from converting an ER diagram to a set of relations and is satisfied if the database created has atomic values in each cell of each table within the database.

The next logical normalization is from First Normal Form (1NF) to Second Normal Form (2NF). A relation is said to be in Second Normal Form if it is already in First Normal Form **AND** the non-key attribute(s) of the relation are derivable only by using the **entire** primary key of the relation.

However, this is not necessarily a required condition as a relation in First Normal Form can also be in Second Normal Form if the primary key of the relation being normalized is a **single attribute**.

The last and most rigorous normal form is Third Normal Form (3NF). A relation is in 3NF if it is already in Second Normal Form **AND** there are no transitive dependencies present in the relation. In this case a transitive dependency **does not** relate to the transitivity axiom for functional dependencies and is instead defined as any relationship between two or more attributes, of which none are part of the primary key of the relation containing them. **The quickest way to convert from Second Normal Form to Third Normal Form is to eliminate transitive dependencies by creating a relation against the transitively dependent attributes and leaving the primary key of the created relation in the original relation as a foreign key reference.**

Key Vocabulary

Term	Definition
First Normal Form (1NF)	The database has atomic values in **every** cell of **every** table within the database
Second Normal Form (2NF)	The database must be in 1NF **AND** the non-key attribute(s) of each relation must solely be derivable using the **entire primary key**
Third Normal Form (3NF)	The database must be in 2NF **AND** there must be no **transitive dependencies** present in the relation.

Topic 5: SQL Syntax

Subtopic 5.1: Creating Tables

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

Subtopic 5.2: Insertion, Update, and Delete Operations

Insert

```
INSERT INTO table_name ( column_name1, column_name2, ...) VALUES ( value1, value2, ...);
```

The list of column names and attributes must be comma-delimited to allow for matching. If a column is named, but no value is provided for that column then the value will be filled in as either NULL or the default defined when the table was created.

Update

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

The **where** condition in the update clause specifies a value or range of values to look for when choosing whether or not to update records in the database. Omission of the where clause will result in updating **ALL** records in the database.

Delete

```
DELETE FROM table_name WHERE condition;
```

The **where** condition in the delete clause specifies a value or range of values to look for when choosing whether or not to delete records in the database. Omission of the where clause will result in deletion of **ALL** records in the database.

Subtopic 5.3: Cascade, Set Null, and Restrict

Cascade

In a **CASCADE** operation, the action is performed in the table containing the primary key and changes are made in the table containing the foreign key so that there is no violation of the foreign key constraint. The cascade operation can be of two types, either an UPDATE CASCADE in which foreign key values are changed or a DELETE CASCADE in which rows with foreign key values in violation are deleted.

Set Null

In a **SET NULL** operation, just as in the cascade operation, the action is performed in the table containing the primary key and changes are made in the table containing the foreign key so that

there is no violation of the foreign key constraint. However, unlike the cascade operation, a set null operation has only one outcome. On either an UPDATE or a DELETE, the foreign key values are set to null.

Subtopic 5.4: Auto-Increment

If we are inserting values into a table in a database, often those values will be associated with some form of Identifier. However, if a dataset does not have an identifier or a meaningful identifier is not easily inferable, then the auto-increment SQL property helps to assign each record an ID. The first record inserted is given ID 1, the second record is given ID 2, and so on.

Subtopic 5.5: Insert Date

If we want to keep information on the Date/Time of objects that we store in the database, then the insert date condition in SQL will help with that. There are two ways to do so: if you want to include the current timestamp, use the SELECT NOW condition. However, if you do not want to include the current timestamp, use the SELECT CURRENT_DATE condition.

Subtopic 5.6: Alter Table

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table and is also used to add and drop various constraints on an existing table.

Add: Add a column

```
ALTER TABLE table_name
ADD column_name datatype
```

Drop: Delete a Column

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

Modify: Change column Data Type

```
ALTER TABLE table_name
MODIFY COLUMN column_name datatype;
```

Modify: Change column Size

```
ALTER TABLE table_name
MODIFY COLUMN column_name size;
```

Subtopic 5.7: Select

The SELECT statement is used to select data from a database. The data returned is stored in a result table, called the result-set.

Select: Specific Records

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Select: All Records

```
SELECT * FROM table_name;
```

Select: Aggregate Function List

AND: Attach clauses together, both must be true for a record to be selected

AS: Rename the resulting column(s) or set of records

COMPARISON OPERATORS: Used for numeric data in the where condition

IN: Select only records containing data **exactly** matching one of the values in the list of values that follows

NOT IN: Select only records containing data matching **none** of the values in the list of values that follows

BETWEEN: Select only records containing data **within the range (including endpoints)** of the values in the list of values that follows

NOT BETWEEN: Select only records containing data **within the range (NOT including endpoints)** of the values in the list of values that follows

LIKE: Select records with data in a particular field matching the specified pattern.

%: Match any number of characters

_ : match **EXACTLY one** character

BINARY: Used in SELECT command to make any strings chosen as comparators **case insensitive.**

The entire library of common string functions (length, upper, lower, etc.) can all be used in select statements, just with specific syntax. Please consult documentation to find the syntax needed.

OR: Attach clauses together, at least one must be true for a record to be selected

NOT: Attach clauses together, whatever follows the NOT must be false for a record to be selected

MATHEMATICAL OPERATORS: : Used for numeric data in the Select condition

FORMAT: Format numbers and data by specifying two things: **the number** and **the number of places following the decimal point**.

DATE: used to select records based on timestamp and can be operated on (Add/Subtract)

NULL/NOT NULL: retrieves only those records whose value for a certain column is (not) null

AGGREGATE FUNCTIONS: Used to gather statistical information about the database such as the number of records (COUNT), the sum of all record values for a column (SUM), the average value for a column (AVG), the minimum value for a column (MIN), and the maximum value for a column (MAX).