2024

CMSC424: Database Design

FINAL EXAM 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	$\text{If } B \in A, \text{then } A \rightarrow B$
Augmentation Axiom	$\text{If } A \rightarrow B, \text{then } AC \rightarrow BC$ **For any C such that $C \neq A$ and $C \neq B$**
Transitivity Axiom	$\text{If } A \rightarrow B \text{ and } B \rightarrow C, \text{then } A \rightarrow C$
Union Rule	$\text{If } A \rightarrow B \text{ and } A \rightarrow C, \text{then } A \rightarrow BC$
Decomposition Rule	$\text{If } A \rightarrow BC, \text{then } A \rightarrow B \text{ and } A \rightarrow C$
Pseudo-Transitivity Rule	$\text{If } A \rightarrow B \text{ and } BC \rightarrow D, \text{then } AC \rightarrow D$
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).

Subtopic 5.8: Regular Expressions

Both MySQL and MariaDB support the use of regular expressions in SQL queries. The keyword to use Regular Expressions is **REGEXP.** However, just as with any other keyword in MYSQL or MariaDB, this keyword is **not** case-sensitive (regexp, Regexp, etc. are all acceptable).

The idea behind the REGEXP command is somewhat similar to a typical programming language, meaning that there are tricks and specifiers that can increase the power of the query being used.

Here are some common ones:

'<character(s)>' → Elements selected contain the character(s) specified in the columns being selected from

[ ] → This is used to define a **set** of characters or strings to match when selecting records from the table

BINARY → The default for any use of the **regexp** keyword is case-insensitive. This makes it **SENSITIVE TO CASE**.

[^] → This is used to define a **set** of characters or strings to avoid when selecting records from the table

Just as in most programming languages, any regexp string can have various constraints. To define these constraints, there are common syntax used in regular expressions that tend to remain constant across languages. The following is a list of the most common specifiers used:

^ → Match at beginning (**DO NOT CONFUSE WITH [^], WHICH MEANS NOT IN SET**)

$ → Match at End

. → Match any **single** character

*→Match 0 or more of **any** character

? → Match 0 or 1 of **any** character

+ → Match 1 or more of **any** character

<character(s)>|<character(s)> → Match **either** left or right

{<number>} → The **exact** number of matches for the character(s) that precede the braces

{<start>, <end>} → The number of matches for the character(s) that precede the braces must strictly be **between** start and end (both included)

[[:alpha]] → matches **any** alphabet character (Uppercase and Lowercase a-z)

[[:digit]] → matches **any** digit character (0-9)

[[:character_class]] → matches **any** instance of a character class (digit, alpha, upper, lower, space, blank, alnum, etc.)

Subtopic 5.9: Group By

In SQL, Group By is used to return a listing of aggregated values. The SELECT statement containing the Group By clause contains at least one **regular column** and at least one **aggregation of a regular column**. The Group By clause runs the Aggregate function on each different value of the regular column, returning the result as a table. If you mix regular columns and aggregated columns, you must have a **Group By** clause otherwise the query will have a logic error. If you want to use a WHERE clause in the SELECT statement containing the GROUP BY clause, then the WHERE clause must come **before** the group by clause. Furthermore, a group by clause can specify more than one column, leading to the formation of what are called **subgroups**. In addition, there is no limit on the number of aggregate functions that can appear in the SELECT clause as the clause can have **multiple** aggregate functions for a single group by statement. Just as Group By is an aggregate function, so too is HAVING. Just as WHERE applies to SELECT clauses, HAVING applies to Group By clauses. The HAVING clause may use an aggregate operation and there is no requirement that the column used in the HAVING clause and the SELECT clause must be the same. Furthermore, Where Group By, and Having can all be used together in a select statement with no ill effects.

Subtopic 5.10: Select – Joins

Sometimes the information we want to return as the result of a query spans across multiple tables. In the event that we need more than one table, the first strategy that we can use is to perform a Join on the tables that we need. Joins combine **multiple physical tables** into a **single logical table**. In practice, the **where** clause is used to perform the join, usually by making use of equal values in corresponding columns (foreign key/primary key). In order to indicate that a join is to take place, the tables to be joined must **all** be indicated in the from clause. In addition, any column names that are ambiguous across the tables being joined **must be qualified with the table name of the table they are from**. In the event that the selections made must be from more than two tables, the tables must be joined to each other based on common columns and taken **2 at a time** until all tables have been joined. If many tables are being joined and each join requires the tables to be taken **2 at a time,** then the table name can be abbreviated using an alias that can then be used in any join or condition statement within the query. Most kinds of joins eliminate the values that are not common to the two tables. However, by using an outer join, these values can be preserved. The order of the tables determines the direction of the join as going from A to B is a **left outer join,** while the reverse direction is a **right outer join**. To perform a natural join in MySQL, use the "**=**" syntax and for outer joins, use the keyword **ON**.

Subtopic 5.11: Select – Nested Queries

Recall that we can use Joins to retrieve information if the query for that information spans across multiple tables. The alternative strategy for querying information from multiple tables is through the use of Nested Queries. The way in which this works is by splitting the information requested into two or more simple queries, which are then nested inside each other. Often times, if a nested query is used, the nesting happens as a part of the where clause in the outer query. This strategy does often work, but often runs into a problem in the event that the inner query does not return a single value. In this case, the issue can be remedied by using the keyword **IN** to indicate that there may be multiple results returned by the query. The **NOT** keyword can be added in front of **IN** in order to invert the results of a query and check that a value is not a member of the result set as opposed to the usual method in which values are checked **for** membership in a result set. If the resulting query is to be limited to a certain number of results, the **limit** keyword can be used to indicate which row to start with and how many rows to return as a result. Due to complexity the general consensus is that the EXISTS keyword should be avoided as should correlated queries.

Topic 6: PHP

Subtopic 6.1: PHP Intro

PHP is a scripting language that is commonly used by web servers to send and receive data from HTML forms. The 'echo' function in PHP serves to print things to the screen. In addition, any special characters must be escaped using a backslash (\). PHP code appears in the PHP tag (<?PHP?>) and can be used as a standalone file or in conjunction with HTML code. Comments in PHP mirror the syntaxes found in JavaScript/C++ (// => single line; /**/ => multi-line) and Python (#)

Subtopic 6.2: PHP Variables

Just as in any coding language, there is a particular syntax to follow for the variable names in PHP. PHP, just like Python is a dynamically typed language that **does not require** declarations of variables. In PHP, variable names begin with a dollar sign ($), followed by any number of Letters, underscores, or digits. When trying to use a variable in an echo statement, the string must be in double quotes ("") for the variable to be interpreted as its value when the echo prints to the screen. Variable names are case-sensitive, so First and first are two different variables.

Subtopic 6.3: PHP and SQL

Often times when connecting to and using a database becomes too difficult, we can use PHP to execute SQL queries for us. Since the database that we have been using for the majority of the class is a MySQL database, the way in which we will connect to it from PHP is using the mysqli class. The class is designed such that it returns a connection resource to the database if successful. Mysqli is usually invoked using its constructor, which has 4 parameters: **the server, the username for the server, the password associated with the provided username,** and **the database to be used on the MySQL server.** To execute SQL queries in PHP, use the mysqli command to make a MySQL database "reference" and then call the query method of the mysqli class.

```php
//file containing necessary constants, name: constants.php
<?php
    class constants{
        const LOGIN = "testuser";
        const PASSWORD = "testpasswd";
        const DBNAME = "cmsc424s24_class";
    }
?>
```

```php
//file showing simple mysql connection, name: php_sql.php
<?php
    include("constants.php");
    $login = constants::LOGIN
    $password = constants::PASSWORD
    $dbname = constants::DBNAME

    $mysqli = new mysqli("localhost", $login, $password, $dbname)
?>
```

Subtopic 6.4: PHP Forms

In HTML, user input is done through forms (<form></form>). In addition, forms can contain other HTML widgets (text fields, radio buttons, dropdowns, etc.) within them. In the event that the user submits a form, we want some way to capture the information that was submitted. To do so, we specify the method of collection (Get vs Post) and send the form data to a PHP script. When processing form input in PHP, there are a few predefined variables, called **superglobal variables**, that help in processing. These include: **$GLOBALS, $_SERVER, $_GET, $_POST, $_COOKIE, $_ENV,** and **$_REQUEST**. Depending upon the method of collection specified in the form (get vs post), the superglobal variable used will vary. If the method was specified to be post, then the retrieval will be done using the syntax **$_POST[variable]**, and likewise if the method was specified to be get, then the retrieval will be done using the syntax **$_GET[variable]**.

```
<!--HTML form for greeting-->
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Greeting Form</title>
</head>
<body>
    <form action="search.php" method="post">
        <p>Enter your name:
            <input type="text" name="name"/></p>
        <p>Enter your age:
            <input type="text" name="age"/></p>
        <p><input type="submit"/></p>
    </form>
</body>
</html>
```

```
<!--PHP response to print greeting to the screen-->
<html>
<head>
    <title>Form answer</title>
</head>
<body>
    Hi <?php echo $_POST['name']; ?>.
    You are <?php echo $_POST['age']; ?> years old.
</body>
</html>
```

A word of warning: using “get” for the method of collection means that the data passed will be in the URL/resource of the PHP file directly. This is generally not advised if the information being passed is meant to be kept confidential.

Subtopic 6.5: PHP Templates

Often times when designing a website, multiple pages may have a common look or feel. In such a scenario, we can make an HTML (or PHP) template and place the template in a PHP file. The syntax to include an external file within a PHP file is **include(“filename”).** In addition, we can set up templates that include variables whose value will be included at run-time and set up the variable to be evaluated inside the template file. If the file inside which the variables are to be evaluated is an HTML file, the syntax is **<?=$variable?>**, meaning that the file will be mostly HTML but with a few PHP expressions in it. In order to ensure that the evaluation happens correctly, the

HTML file must be included in the PHP file with the variables, but the include statement must come **after** the variables have been declared and assigned.

Subtopic 6.6: PHP User-Defined Functions

PHP's ability to interface with SQL and HTML means that there are many inbuilt functions in PHP such as count, include, and array functions (array_pop, array_push, etc.). However, just as in any other coding language, which PHP is one of, there can be user-defined functions. User-defined functions begin with the **function** keyword and must contain **only valid PHP code**. Like most other coding languages, any parameters declared in the function header are **passed by value**. In addition, values are returned from functions using the return statement and there is no restriction on the return type of the data from the function. Functions in PHP can be defined to use **pass by reference** as opposed to **pass by value** for a particular parameter if the chosen parameter is preceded by an ampersand (**&**). In addition, like some other languages such as JavaScript, PHP does support a variable-length argument list and the functions **func_num_args(), func_get_arg(),** and **func_get_args()** can be used to retrieve the arguments.

func_num_args() => return number of arguments provided in argument list of variable length

func_get_arg(n) => provide argument #n from the argument list of variable length

func_get_args() => provide an array in which each element is the argument corresponding to that number in the argument list (get_args[0] = 1st argument, get_args[1] = 2nd argument, etc.)

Subtopic 6.7: PHP Frameworks

When a database is created, it often has multiple tables encompassing different datasets that we would like to keep track of. If we want to convert the database to a digital format, the best means for doing so is to build various classes in PHP. As a rule of thumb when converting a database to PHP, each of the relations in the original database (each table) will become a class of its own in the PHP. Frameworks are a general software concept and are **not** unique to PHP. Generally, a framework is considered to be a software program that, at the very least, attempts to create a base class for every table in a database. These frameworks or base classes can then be extended in order to add more functionality as desired. Just like many scripting languages, PHP does follow the Model-View-Controller architecture. In PHP, classes are the model, with HTML/JS as the view, and other PHP scripts as the controller. Some well known PHP frameworks are **Symfony, CakePHP, and Laravel.**

Topic 7: Cybersecurity

Subtopic 7.1: SQL Injection

Before we can discuss SQL injection, we need to understand Databases themselves. This is because at the heart of every query is a database lookup. Databases are made up of named tables that are defined by their columns (attributes) and rows (records). The unit of work on a database is what is called a transaction. Transactions, at least for the sake of security, should be ACID. ACID here is an acronym for Atomicity (transactions are either done fully or not done at all), Consistency (The database is not made invalid by the execution of a given transaction), Isolation (the resulting database following a transaction is not visible until the transaction has been completed), Durability (the completion of a transaction should be maintained in a database despite external adverse events). In order to query databases, there must be a standard query language defined. This is called SQL (Structured Query Language). Typical SQL statements include keywords indicating transactions (SELECT, UPDATE, DROP, etc.), keywords indicating table information (FROM), and keywords indicating conditions to be met by the data being selected or changed (WHERE, LIKE, etc.). All SQL statements must be followed by a semicolon (;), but some may also be followed by an additional comment (indicated with " -- ").

Now that we have a basic knowledge of databases and how to query them, we can start to explore the security vulnerabilities of using databases. Let's take a hypothetical example. Let's say that we have a website (we built it for testing) where we have some login code in a PHP file. We also know that this code queries our database of users and logs in if the query returns any rows at all. This simple fact can be exploited if the login fields are not sanitized. For example, if the following username were entered: "frank' OR 1=1); -- ", then the website can be used without entering a password as the login query now becomes: "Select * from Users where (name = 'frank' OR 1=1); -- ". In this case, the 1=1 will always evaluate to true, so regardless of what the first half of the where clause returns, this query will always allow somebody to log in to the website. This exploitation of a database using SQL and conditional logic is just one of the many different types of attacks under the umbrella of SQL Injection.

Just as with any attack, there are countermeasures that can be taken. The simplest and most naïve approach to preventing SQL injection is to blacklist the problematic characters (', --, ;). However, this raises the issue of users with names containing those characters (O'Neil, O'Connor, etc.) which means you do sometimes want those characters. This makes it difficult to decide if/when the characters are bad. Another approach to preventing SQL Injection that does better than blacklisting but may still not be perfect is Whitelisting. In Whitelisting, the database operator checks that the user input is in some set of values known to be safe (an integer in the right range for example). Yet another approach to preventing SQL Injection that works better, but not perfectly, is escaping characters that could alter control ("\' , \; , \- , \\). However, this runs into the same issue as blacklisting since these characters may be needed at times. The strongest defense against SQL Injection, and one that works well if executed correctly, is to use prepared statements and bind the data later on. Essentially what this does is it forces the inputs to be of a certain type, all but eliminating the chance of SQL Injection (however the chance is still non-zero). Noting that there is no perfect solution to SQL Injection attacks, the best efforts can be made to mitigate the impact

should such attacks take place. The best way to do so is to limit privileges (the commands/tables a user has access to) and to encrypt sensitive data stored anywhere in the database.

Subtopic 7.2: One-Way Encryption

Let's assume we have a database that is for users who are logging in to a website that we run. In this database, let's now assume that we are storing the user's information (username, email, password, etc.). The storage of passwords in the database necessitates encryption. The best kind of encryption to use in this scenario is what is known as a **one-way safe encryption scheme.** In a one-way encryption scheme, a plaintext value can be encrypted but cannot be later decrypted.

Topic 8: SQL and Java

Subtopic 8.1: SQL in Java

In Java, the java.sql package provides classes and interfaces for connecting to a RDBMS and executing SQL queries. The collection of classes within java.sql are known as the JDBC (**J**ava **D**ata**b**ase **C**onnectivity) classes. Among these JDBC classes are some important ones such as **DriverManager**, **Connection**, **Statement**, **ResultSet**, and **SQLException**. The syntax for connecting to a database is almost similar to that in PHP whereby a method is used for connection, however unlike PHP, the parameter to the method is a URL as opposed to 4 separate parameters representing the identifying information. If the connection is successful, a connection object reference is created, and from this reference, a statement object reference can be obtained. It is the obtained statement reference that can then be used to perform typical SQL queries such as Select, Update, and Delete. If the query initiated through a statement object reference is successful, the result is returned as a ResultSet which can then be looped over in order to access the values of the columns for each row of the result set. In order to aid with accessing values and understanding the returned results, the ResultSet interface has useful methods such as **next, getString, getInt, getDate** and more.

Topic 9: NoSQL

Subtopic 9.1: NoSQL Intro

Recall that typical SQL tables store atomic data at the intersection of each row and column. With the advent of NoSQL, we can move away from the atomicity restriction of SQL and can store any kind of data in a NoSQL table (threads on a discussion post, XML documents, JSON objects, etc.). The main reason for the rise of NoSQL databases was the growth in the amount of data which raised questions about the performance of SQL. Some of the common NoSQL databases are **MongoDB**, **Redis**, **Cassandra**, **HBase**, and **Neo4j**. There are different types of NoSQL databases. Some databases are key-value stores (hashtable), some use wide-columns as opposed to rows, some like MongoDB are document databases (keys map to documents), and some are graph stores (used to store networks). The main idea behind using NoSQL as opposed to traditional SQL is that many scenarios do not require the full set of features provided by SQL but do still require the speed of access that SQL provides. Some of the key needs such as insertion at the end or selection of multiple elements can be done better and more efficiently in NoSQL especially for larger volumes of data. In addition, another advantage of NoSQL databases as compared to their SQL counterparts is that a NoSQL database does **not** need to know the structure of the data it is storing beforehand. Because of the dynamic nature of NoSQL databases, JSON strings are most often used to store data in a NoSQL database. Furthermore, the use of JSON to store data in a NoSQL database allows the database to be widely used as most common programming languages have a function or library dedicated to parsing JSON. Despite the multiple references to the lack of structure in NoSQL databases, the JSONs used do have an internal structure. To convert an SQL table into a JSON, any rows in the SQL table become objects in the JSON whilst columns in the SQL table become keys for the values in the JSON.

Subtopic 9.2: MongoDB Intro

MongoDB is a NoSQL database that is open source, document-based, and written in C++. MongoDB can have multiple databases, which are sets of collections. A collection in MongoDB is a set of documents and serves as the equivalent of a table in SQL. Documents in a collection are often of a similar nature but may contain different fields within them. At their simplest, documents are just a set of key-value pairs that support a dynamic schema since the data to be inserted is often not known beforehand. As documents in MongoDB cannot be identified uniquely by the data they contain, each document is assigned an ID. This ID is stored in the variable _id and it is the key of a document, meaning that each document has a unique value for the variable. This value can be user-defined if it is provided when inserting but can also be auto-generated by MongoDB if it is not provided when inserting (a little bit like auto_increment in SQL). MongoDB, just like any other software, has a unique syntax.

SYNTAX TO REMEMBER

1. To drop a collection, use the **drop** command
2. To drop a database, use the **dropDatabase** command
3. To make a collection (table), use the **createCollection** command

    a. If a collection is specified as capped, we need to specify the maximum allowable size for the collection at the time of creation.
4. To insert documents into a collection, use either the **insert** or **save** commands
    a. If the id of the document being inserted is already present in the collection, MongoDB will throw an error.
    b. Insert has two versions: **insertOne** (single record) or **insertMany** (multiple records)
    c. If an id is not specified in the save command, then the functionality is identical to insertOne.
5. There are two methods to find documents in a collection: **findOne** (single document) or **findMany** (multiple documents)
    a. The find command can be used with all columns and common operators in order to use it sort of as the "Where" clause in an SQL query.
    b. The find command also supports regular expressions, which must appear between backslashes (/). (Example: {"title": /[ei]/} => document title contains e **or** i)
    c. Comparison operators:
        i. Less than => $lt
        ii. Greater than => $gt
        iii. Not Equal => $ne
        iv. Less than equal => $lte
        v. Greater than equal => $gte
    d. The find command also supports logical operands such as OR, AND, and NOT.
6. While the save command saves documents within a collection, the **updateOne** and **updateMany** commands save data within a document.
    a. Both update commands must be used with the set method (**$set**).
7. To remove documents from a MongoDB collection, use the **deleteOne** (single document) and **deleteMany** (multiple documents) commands.
8. A MongoDB collection can be sorted by using the **sort** command
    a. The sort **must be** on a find object, so it is usually chained to the end of a find
    b. The sort direction is specified by a key, with **ascending as +1** and **descending as -1**
    c. A comma separated list of fields can be used to specify multiple fields on which to sort the results of a find.
9. MongoDB supports all common aggregate operations through use of the **aggregate** command.
10. An index (for easy access) can be created on the members of a MongoDB collection using the **ensureIndex** command.

Subtopic 9.3: MongoDB Pipelining

When performing aggregate operations, we could either use single purpose aggregation methods (which can quickly become tedious) or we could use pipelining operations (which are slightly more complicated but are more powerful). One of the most common aggregate operations is a count of the number of documents in a collection. In MongoDB, this is done through the **countDocuments** command in which any needed conditions can also be specified. Another common aggregation is counting the number of unique values of a field. In MongoDB, this is done using the **distinct** command, with the desired field passed in as a parameter. These are simple

tasks, but they can become tedious in the event that multiple are executed in quick succession. The MongoDB alternative is to use what is known as pipelining, a process of using aggregate operations that occur in various stages with each subsequent stage depending on the result from the stage before it. The stages in the MongoDB pipeline can largely be broken up into four major types: **Filtering, Projecting, Grouping,** and **Sorting.** When pipelining in MongoDB, it is key to remember that variables and field names are preceded by a dollar sign (**$**) while regular strings are **not**. The four types of stages in a MongoDB pipeline contain many possible methods within them. Some methods of note are: **$match, $group, $bucket, $project, $skip, $replaceWith, $sample, $facet, $lookup, $unionWith, $out,** and **$merge**. The main purpose of using pipelining is to optimize any queries made on a MongoDB collection (by, for example, reducing the number of documents present in each subsequent stage).

Subtopic 9.4: PHP and MongoDB

Just as with MySQL, we can use PHP to connect to MongoDB. However, the classes required to connect with MongoDB do not come installed in the standard version of PHP, meaning that they must be specially installed. MongoDB\Driver\Manager is the main entry point when connecting from PHP to MongoDB and this class has multiple methods for reading and writing from a MongoDB instance. To write to a MongoDB database, we use the executeBulkWrite method in the driver manager class which allows us to execute one or more write operations. Just as there was a way to perform queries using PHP when working with MySQL, there is an analogous method for MongoDB called executeQuery. The method has one parameter which is an array representing the filters for the query.

Topic 10: Implementation and Query Processing

Subtopic 10.1: Implementation and B+ Trees

To look up a record, we must first look up the index for the desired record (this can be thought of like searching for a word in the index of a book). After the desired index is found, we can go to that index and will likely encounter a pointer to the record with which we can retrieve the record (this can be thought of as finding the page number(s) where the desired word appears). With regards to databases, there are two main kinds of indices: **Ordered indices** (the values are in some form of a sorted order) and **Hash indices** (the values are stored in buckets and a hash function will find the appropriate bucket given an index). If the index being used is what is known as a Primary Dense Index, every key **must** appear in the index. As such, the index can get quite large but does prevent costly accesses to records directly. On the other hand, however, if the index being used is still a primary index but is sparse as opposed to dense, then the index may fit in memory better, but the tradeoff is that due to a lack of records in the index, each query **must** access the relation, which can end up becoming costly.

An index is always **automatically** set up for the primary key attribute of any table or relation. However, we can also manually set up an index on the value of a non-primary attribute if we have reason to believe that multiple operations will happen based on the values associated with that attribute. However, unlike the possibility of a sparse index on a primary key, since the secondary index is not based on a primary key, the secondary index must be dense so that some search query can find every record in the table.

However, using indices can quickly become tedious as insertion and deletion require the data and the index to be updated. For this reason, some indices can be implemented as what are known as B+ trees. These trees are Binary trees that self-balance, meaning that they always remain balanced and the length of the path from the root of the tree to any leaf of the tree will always have the same length, regardless of insertion or deletion. Nodes within the tree contain pointers and values. The number of pointers or values in a node can be used to separate nodes into three types: Root, Internal, and Leaf. Internal nodes contain both values and pointers. They also have a constraint on the minimum and maximum number of pointers and values that they can contain. The root node usually has the same number of values as the internal nodes of the tree, but can at times be smaller if the entire tree is very small (empty or a few values). All internal nodes of the B+ tree that are **not** the root have between $\left\lfloor \frac{n+1}{2} \right\rfloor$ and n pointers. Furthermore, these nodes have a number of values equal to their number of pointers -1 (for example: if an internal node has 7 pointers, it will have 6 values). A typical leaf node is a list of primary key values and their associated pointers. To find a value in a tree, when searching we end up in a leaf that contains the appropriate primary key value that can lead us to the corresponding row in the table from which the tree was made. The leaves typically have pointers based on the index values from the index created on the table from which the tree was made. However, depending on the type of index used, the values associated with the pointers will vary. If a primary index was used, the value for each pointer is the records themselves, but if a secondary index was used then the value for each pointer is a bucket of records that contains a pointer to the records contained therein. A B+ tree is defined by its order, n. With regards to the tree, n defines the number of children each node has, so a larger n results in a flatter tree.

To update based on **a primary key value** in a B+ tree, we search the B+ Tree for the leaf node containing that primary key value, access the corresponding record/row, and update it. To delete from a B+ Tree based on **a primary key value** we search for the node containing that particular value, delete the corresponding record (or row) in the table, and rearrange the B+ tree to maintain balance as needed. As the tree is self-balancing, the runtime is guaranteed to be $\log_n(\#\ of\ rows)$ provided that the selection criteria for the search is based on **a primary key value**.

To perform insertion on a B+ tree, start by traversing from the root to find the appropriate leaf node for insertion, adhering to the tree's key structure. If the leaf node has space, simply insert the new key-value pair while maintaining the node's sorted order. However, if the leaf node is full, initiate a split operation to accommodate the new entry. During the split, redistribute the keys evenly between two nodes, ensuring the median key moves up to the parent node. This process may propagate upwards, potentially necessitating further splits until the tree's balance is restored. Finally, update the parent pointers accordingly. Through this methodical approach, the B+ tree maintains its properties of balanced height and efficient search, while accommodating new data seamlessly.

On the other hand, when performing deletion on a B+ tree, begin by locating the target key within the leaf nodes through a traversal from the root, maintaining adherence to the tree's key structure. Once found, remove the key-value pair from the leaf node. If the removal doesn't violate the minimum occupancy constraint, the operation is complete. However, if the removal causes the node to fall below the minimum occupancy, initiate a redistribution or merge operation with neighboring nodes. Redistribution involves borrowing a key from a neighboring node or merging two nodes into one if possible. Ensure to update parent nodes accordingly to maintain the B+ tree's integrity and balance. This process might recursively propagate up the tree until the root node is reached, ensuring the tree remains balanced and efficient even after deletion.

Subtopic 10.2: Query Processing: Intro and Joins

Every query has multiple ways to execute them. Since queries are nothing more than a sequence of operations, the order in which those operations are executed can be permuted without any ill-effects on the result of the query. One operation represented using linear algebra is called an **evaluation primitive** and the sequence in which these operations are combined to make a query is called a **query execution plan** or a **query evaluation plan.** Given an SQL query, the associated RDBMS being used will try to construct a query evaluation plan that minimizes the **cost (evaluation time)** of the query. Every major kind of query (Select, Insert, Update, Delete, etc.) is broken down into a query execution plan before the RDBMS actually executes it.

The other major thing that can be handled by query processing is the sorting of records. Let's assume, for simplicity, that the entirety of a relation does not fit into memory. In order to make sure that the sorted relation does fit into memory, we will employ the **external sort-merge algorithm** with a baseline value **M** representing the number of disk blocks available in memory. This process has two phases. In the first phase, the relation is broken up into several pieces (each of which fit in memory) and then each piece is sorted before the memory writes it to disk and clears it. Each of these portions of memory are called a **run** and the second phase merges the runs to output the sorted relation. Often times we want to leave space for output (usually 1 block of memory). This

leaves us with only M-1 blocks available for data, so the number of runs must be reduced down to M-1 in order to be able to begin the second phase of the external sort-merge algorithm. In the second phase, runs are taken in groups of size M-1 and are merged into runs until the number of merged runs is less than or equal to M-1.

Just as external sort-merge is a type of join, there are also other joins. The first and simplest join is called **Equi-Join or Natural Join** and it can be summarized as follows: If there are two tables **R** and **S**, then the Equi-Join or Natural Join of R and S is the process of taking **each tuple in R** and finding **all matching tuples in S.** The second kind of join is a **Nested Loop Join** and it functions exactly as the name implies. **It is a double loop over all records in both R and S to check if the records match on the value of the desired column or columns**. The slightly more optimal version of the nested loop join is called the **indexed nested loop join** and it modifies the inner loop of the nested loop join by introducing an index that reduces the number of comparisons made in the table that has been indexed. The third unique type of join, and arguably also the most complex, is the **merge-join (or sort-merge join)**. This join uses the sort-merge algorithm to join its records. If we have two tables R and S, two pointers are created p_r for table R and p_s for table s. The values at the two pointers are compared and if they are equal then the tuples are output and p_s is moved down. If the two values are unequal, then the pointer to the smaller value is moved down. Comparisons between the values at the two pointers continue until the end of one file (Relation) has been reached. The last kind of join is arguably the fastest. It is called a **hash join** and given two relations R and S; it uses a hash function to partition one of the relations such that the relation fits in memory. It then uses a nested loop to read the other relation block by block and find matches for its values in the hashed relation, repeating until the larger relation has been entirely consumed.

Most of the previous joins mentioned are inner joins of sorts, since they discard all tuples that do not match between R and S. However, SQL does also have outer joins. In a left outer join of two relations R and S, the tuples that remain are all from R while only the matching records from S are maintained. In a right outer join of two relations R and S, the tuples that remain are all from S while only the matching records from R are maintained.

Subtopic 10.3: Query Processing: Materialization and Pipelining

Query Processing depends on the order and speed with which the constituent parts of a query are processed. Just as Joins help with the data, evaluation of expressions is helped by Materialization and pipelining. In the process of **materialization,** every expression within a query is evaluated separately with the results stored in temporary relations that are then accessed for subsequent operations. This serves to prevent repeated accesses to the disk, which can quickly become costly and slow. The alternative to materialization is **pipelining** whereby multiple operators are evaluated simultaneously. Usually this is faster since this skips the disk access entirely but is not often possible due to memory or availability constraints. Because of memory constraints, sorting and aggregation operations **cannot** be pipelined, but the second longer phase of algorithms such as sort-merge and hash join can be pipelined.

Subtopic 10.3: Query Processing: Query Optimization

Recall that every SQL query has multiple ways in which it can be executed and often those methods of execution vary in both cost and time taken. In addition, there are many other choices. For example, with queries involving multiple tables, the join order becomes a choice although it often does not have much impact. In an ideal scenario, the best plan would be to generate all possible evaluation plans for a particular query, evaluate each of those plans by cost, and then choose the best. However, in a real world scenario, there may be too many different execution plans possible for a single query. In such a scenario, we likely settle for a plan that is "good" but not objectively the best. Typically, the entire search space is usually interleaved into a single efficient search algorithm. With regards to expressions, two expressions are considered to be equal if and only if their results are identical on all legal databases. If given an expression, to generate all possible equivalent expressions, begin with the original expression and apply all possible rules recursively until no new expressions are generated by application of ANY particular rule. The algorithms used for optimization can largely be split into two groups: Exhaustive (traverse the whole search space and find the **optimal** solution) and Heurisitical (Simpler, but not always guaranteed to find the optimal solution).

Subtopic 10.4: Query Processing: Concurrency and Deadlock

Subtopic 10.5: Association Rule Mining

Association Rule Mining is the process of looking at historical data to suggest other products or services that may interest a user. At the heart of Association Rule Mining are rules. Rules appear similar to functional dependencies but have different terminology for the parts of the rule. The left hand side attribute(s) of a rule are called the antecedent(s) and the right hand side attribute(s) are called the consequent(s). A rule has a support of s% if s% of all transactions contain both the antecedent(s) and consequent(s), while a rule has a confidence of c% if c% of all transactions containing the antecedent(s) contained the consequent(s). To conduct Association Rule Mining, we must first find all (frequent) itemsets that have transaction support above minimum support and then use the frequent itemsets to generate the Association rules that have a confidence level above a certain threshold. A clever trick to identify rules that will be part of the ruleset is to remember that any **subset** of a frequent itemset will also be frequent. An itemset having k items can be generated by joining 2 frequent itemsets having k-1 items (JOINING step), and deleting those that contain any subset that is not frequent (PRUNING step).

```
L1 = { frequent 1-itemsets }
k = 2
Keep doing as long as Lk-1 is NOT empty:
 Ck = candidate itemsets of size k generated from (Lk-1);
 Check if the candidate itemsets are frequent; each frequent one is added to Lk add 1 to k
end
At the end, we have generated L1, L2,L3 …
```