Database Management Systems

- An **entity** is anything that we store data about.
- An **attribute** is the data that we store about the entity.
- Eg - If an entity is a person, then their name, username, password, age, phone number, etc are their attributes.
- A **relationship** is a link between attributes and an **entity**.
- A **row** is all the attribute values of a specific **entity**.
- A **column** is all the values of a specific **attribute.**
- An **entity type** is the category of the entities that are being stored. An example of an **entity** would be CalebCurry, BillyJoe, Jake, etc. An example of an **entity type** would be ***user.*** Everyone (CalebCurry, BillyJoe, Jake) stored in the table that contained the attributes (name, username, password) is a ***user***.
- An **attribute type** is the category of the attributes. Eg - username, password, name, etc.

In summary, attributes are the columns in a table and each row in a table contains all the information about a single entity. All the rows in the table contain information about a single entity type (Eg - user). The name of the table is the **entity type**.

DBMS

- A **DBMS** is a system that allows us to query, view and manipulate the data stored in a database.
- A Relational **DBMS** is a type of DBMS that deals with relational data.
- A **view** mechanism allows us to display selected parts of the data from the database. Any change that we make to a table will not be reflected in a **view**. For that change to be reflected, the **view** needs to be updated.
- A **view** also provides a security mechanism as it allows us to provide access to only the particular data that a role needs.
- A **transaction** is a process that is either completely completed, or not at all. Eg - Withdrawing money from an account involves **reading** the current amount, **deducting** the amount to be withdrawn and then updating the current amount in the account. In case any step is interrupted, the entire transaction will be rolled back.
- The RDBMS stores info on the local storage of the system on which it is running, reads the data from it and presents that data as tables.

SQL

- Stands for Structured Query Language.
- It is used by the user to communicate commands to the database.
- It can do two things: Define the database structure through Data Definition Language (DDL) and Manipulate the database contents through the Data Manipulation Language (DML).
- A **join** is a means of combining data in fields from two tables by using values common to each table.

Naming Conventions

- A **naming convention** is a pattern that we adopt while giving names to keep things consistent.

Database Design

- We design databases, so that Data integrity is maintained and Update Anomalies are avoided. Related tables should have consistent data (Eg - phone number of a user should be the same in all tables).
- A schema tells us how the data is structured in a database.
- There are 3 types of schemas in database design: **Conceptual**, **Logical** and **Physical**. The schema type goes from **general** to **specific** in the said order.
- The **Conceptual** schema shows us how the data is related. Here, we speak of entities and how they are related to each other.
- In the **Logical** schema, the entities are defined as full-fledged tables with columns and each column having a datatype. Also, the exact relationship between the tables is defined (Eg - **one** user can have **many** sales so, the relationship between the **users** table and the **sales** table is **one-to-many**).
- The Physical schema involves implementing the database.

Data Integrity

- Data Integrity means having correct data in the database.
- There are 3 types of data integrity: **Entity Integrity**, **Referential Integrity** and **Domain Integrity**.
- We don't want repeating values, incorrect values or broken relationships between tables.
- Let's take 2 tables: **user** and **sale**. A sale takes place when a user buys a product. This means that the user_id of the same user on both the tables must be the same. If the user_id for the same user on both the tables isn't the same, then there is a data integrity issue.
- An id (key) is used to enforce uniqueness among the entities. A unique row in a table is called an **entity**. Ensuring that only unique rows exist in a table is called **Entity Integrity**. Eg - If two users have the same name and age, then we can introduce a user_id in the users table to make each row unique.
- The question of **Referential Integrity** comes into being when we refer to the IDs of 2 tables. Eg - If there's a **comments** table which contains comments made by different users from the **users** table, then each comment must have been made by a user, i.e., the user_id of the user who made a comment must be the same in the **users table** and the **comments** table. It also means that for every comment, there must be a user. If this condition isn't fulfilled, it means that **Referential Integrity** is violated.
- **Domain** is the range of acceptable values that can be stored in a particular column.
- **Domain Integrity** is ensuring that a particular column in a database is storing values within its domain.

Database Terms

- **Data** is anything that we store in a database.
- **Database** is the place where we store our data.
- **Relational Database** stores data in tables.
- **DBMS** is a system used to control a database.
- **Null** is a value that represents no data.
- **Anomalies** are errors within our data integrity. Eg - When we update something, instead of updating 1 column, it updates many.
- **Integrity** is implemented to protect against anomalies.

- An **Entity** is anything that we store in a database. Eg - A user.
- An **Attribute** is the kind of data that we store about an **Entity**. It is the column in a table. Eg - Username of a user.
- A **Relation** is just another word for a table.
- A **Tuple** is another word for a row.
- A **Table** is a physical representation of **Relation**.
- A **File** is another name for a table.
- A **Record** is another name for a row.
- A **Field** is another name for a column.
- A **Value** is the specific information stored in a column. It is also called **Design**.
- **Database Design** is the process of designing our table to remove anomalies and have data integrity.
- **Schema** is the drawn out structure of our database.
- **Normalization** is a series of steps that we need to follow to get the most optimal database design.
- **Naming Convention**
- A **Key** is a field that we use to have unique rows within a table.
- An **Entity** can be a unique row in a table or it can be a table.

Even More Database Terms

- **DDL** stands for Data Definition Language and is used to create and modify the table structures.
- **DML** stands for Data Manipulation Language and is used to manipulate or query the data in the table.
- The **Frontend** is on the client side and is used to provide a way to access particular data from the database to an end user.
- The **Backend** is on the server side and is used to host the database. The Frontend sends requests to the Backend using a Server Side Scripting Language.

Atomic Values

- The values that are stored in the database must be atomic (indivisible, thus 1 thing) in nature. The value stored in the intersection between a row and a column in a relation must be atomic, i.e., a single value.

Relationships

- A **Relationship** describes how any 2 or 3 entities are linked to each other.
- There are 3 kinds of relationships: **One-toOne**, **One-To-Many** and **Many-to-Many**.

One-To-One Relationship

- In this relationship, an entity can be connected only to another entity. Example, A husband and a wife. A husband can have only 1 wife and a wife can have only 1 husband.
- One person has only 1 SSN and that SSN belongs to only 1 person.

One-To-Many Relationship

- In this relationship, an entity A can have a relationship with many entities. However, each of those many entities can have only 1 relationship with entity A.
- A user can have many comments. Each of those comments belong to that particular user.

Many-To-Many Relationship

- In this relationship entities can have N relationships with other entities and each of those entities can have N relationships with the said entities.
- An example would be students having multiple classes and each of those classes having multiple students.
- Many-to-Many relationships can't be directly implemented in a relational database, instead they must be implemented as One-To-One or One-To-Many relationships.

Designing One-to-One Relationships

- One-to-One relationships are often stored as attributes. An example would be the username of a user. Since a user can have only 1 particular username and that said username can belong only to that user, thus it should be stored as an attribute. Eg - **user_id -> username**. Each unique user_id has a unique username.

- There are examples when a 1-1 relationship will be stored over multiple tables. An example would be a situation where a credit card company issues only 1 card to each of its users. This means that each user can have only 1 credit card and only 1 credit card can belong to 1 user. So, we can store the user_id, username, name and credit_card_no in a single table as username, name and credit_card_no are all dependent on user_id. However, if we wanted to store the issue_date and late_fee of the credit card, then those attributes would depend on the credit_card_no. Thus, we should create a separate table where user_id, credit_card_no, issue_date and late_fee are stored as this table is linked to the first table through the user_id and credit_card_no. So, 2 tables **Users**(user_id, name, card_no) and **Card**(card_no, user_id, late_fee).

- Thus, if we want to store extra attributes about a 1-1 relationship, they should be moved to a new table. This is because a table should contain information only about 1 entity and a row should also contain information about 1 entity.

- On an app, if we allow a user to have a relationship (H) with only a single user, then that creates a 1-1 relationship where the said user is in a relationship (H) with that user and vice-versa, then we can have a table that stores that stores user_id, name, phone_no, relationship (H) (the other user's id).

Designing One-to-many Relationships

- In the previous example, we had a 1-1 relationship where 1 user could have 1 credit card, we created 2 tables: **Users**(user_id, name, card_no) and **Card**(card_no, user_id, late_fee) since we needed to store many attributes about the card. Here, we put the primary key of each table into the other as the foreign key and vice-versa.
- However, if we want a 1-M relationship where 1 user can have many cards but each card can have only 1 user, then we need to take the primary key (user_id) from the

Users table [**ONE side**] and put it onto the **Cards** table [**MANY side**]. So, the 2 tables will now be: **Users**(user_id, name) and **Cards**(card_no, user_id, late_fee). The combination of (card_no, user_id, late_fee) will be a unique entity for each card stored for the user.

Parent-Child Relationships

- Keys keep things unique in a table.
- In a One-to-Many relation, there will be a parent table and a child table.
- The primary key is the parent and the foreign key is the child.
- The foreign key is inherited by the child from the parent and points back to the parent.
- In all tables, each row describes a unique entity.
- In a child table, each row describes a unique entity and has a foreign key value that points back to the parent table.
- In a parent - child relationship, the child always inherits the foreign key from the parent and the primary key is always in the parent. Thus in One-to-Many and Many-to-Many relationships, we must decide which tables are parents and which are children.
- Out of context example - An order always needs a parent (A user / customer who has placed it by buying something). It can't exist without a parent.

Designing Many-to-Many Relationships

- Many-to-Many relationships are handled by splitting them into 2 One-to-Many relationships.
- Each of these One-to-Many relationships connects to a junction / intermediary table.
- Example - A class has many students and each student in that class also takes many classes.
- So, in this approach, we can't decide clearly which is the parent class and which is the child class. So, we construct a junction table which contains the primary keys of the **Students** table and the **Classes** table as **foreign keys**.
- So the 3 tables will be **Students**(student_id, first_name, last_name, age), **Classes**(class_id, class_name) and **StudClass**(student_id, class_id).
- Here, the foreign keys point back to their respective tables.

Summary of Relationships

- We must decide what kind of relationship exists between the entities given to us.
- Let us take the example of a professor and a class. The following types of relations are possible among these 2 entities:

- A professor can teach only 1 class.
- A professor can teach many classes.
- Many professors can teach many classes.

- We need to decide the type of relationship depending on the type of situation.
- If 1 professor can teach many classes, then we can have a Many-to-Many relationship where the tables are as follows: **Professors**(prof_id, prof_name); **Classes**(class_id, prof_id, class_name). The primary key of the ONE side becomes the foreign key of the MANY side.

Introduction to Keys

- A key is always unique.
- All attributes in a table depend on the key.
- The key shouldn't be changed. If we have to change it, then it tampers with the integrity of our tables. It also affects the relationships between the tables.
- A key should never be null.
- A key protects our data integrity. Eg -
- A key keeps everything unique. We can't have 2 entities (rows) that're exactly the same.
- Improves functionality of our database.
- Makes updating easier (If we update a value in the parent table, then that value is automatically updated everywhere else).
- Allows for added complexity. Using a lookup table reduces the repeatability of data.

Primary Key Index

- A primary key is the column or columns that contain values that uniquely identify each row in a table.
- The Primary Key serves as the index of a table.

Look Up Table

- A look-up table consists of 2 attributes: One that is a primary key and another that is a string value.
- An example would be Membership types. Let there be a relation **Membership** (membership_id, membership_type).
- There are some advantages in using this approach for membership: It ensures that users can only select a valid membership type, prevents the membership type field from being empty, etc. The biggest advantage is that the membership type can be included in the **Users** table using the membership_id. This requires us to change the membership_type only in the **Membership** table. If we were to include the membership_type attribute directly in the **Users** table, then, if we wanted to replace an existing membership_type with another, we would have to replace all instances of the existing value with the new value in the **Users** table.
- A lookup table is useful for a set number of options, with a key.

Super Key and Candidate Key

- A **Super Key** is any number of columns that forces every row to be unique.
- If a table in our database has non-unique rows, then we need to add a column that makes it unique.
- A **Candidate Key** is the smallest **Super Key**. It is the smallest number of columns that is needed for a row to be unique.
- The **Super Key** asks the question -> **Can Every Row Be Unique?**
- The **Candidate Key** asks the question -> **How Many Columns Are Needed For A Unique Combination?**
- An example of a Super Key is: (**username**, **email**). Here, we have 2 Candidate Keys: One is **username**, the other is **email**. One of them can be chosen to be the Primary Key and the other one will be an **Alternate Key**.
- The **Alternate Key** is a Candidate Key that is not currently being used, but can be a Primary Key.

Primary Key and Alternate Key

- All the CKs not selected as PK are AKs.

Surrogate Key and Natural Key

- Surrogate keys and Natural Keys are categories of primary keys.
- Used for DB design purposes.
- A **Natural Key** is a Primary Key that naturally exists, has real world value (has some practical purpose) and doesn't have to be separately thought of and defined. Eg - **Username** in Users table.
- **Natural Key** is a column that's already in the table, while **Surrogate Key** is a column that we have to add to it as we aren't able to find a **Natural Key**.
- A **Surrogate Key** is a Primary Key that we need to think of and define specifically. Eg - **user_id** in Users table.
- An example of if we want to use **Surrogate Keys** would be if we want to give an **ID** column to each table. If we have 3 tables, **Users, Sales** and **Comments**, then we have to provide a **Surrogate Key** to each of them in the form of an id (**user_id**, **sales_id** and **comment_id**).
- **Surrogate Keys** are always kept private. Users don't know about them.
- We should try to use **Natural Keys** over **Surrogate Keys**.
- We can use only either **Natural Keys** or **Surrogate Keys**. If we have a surrogate key in a table that has a natural key, then the surrogate key is used as the primary key.

Surrogate Key vs. Natural Key

Surrogate	**Natural**
A **Surrogate Key** is a Primary Key that we need to think of and define specifically.	A **Natural Key** is a Primary Key that naturally exists, has real world value (has some practical purpose) and doesn't have to be separately thought of and defined.
Has data that has no real world value and is solely used for the purpose of uniquely identifying a row.	Has data that has real world value. (Eg - (username), (comment, comment_datetime)).
Is always private.	Can be known to users.
Results in a larger DB.	Results in a smaller DB.
Used when a natural key isn't available or is too large (has too many columns).	Used when a good natural key (unique, few number of columns) is available.
Is Never Changing.	Can sometimes change (And this will require us to update all instances of the changed key value everywhere).

Foreign Keys

- It references a **Primary Key** in the **same table** or **another table**.
- A table can have multiple foreign keys.

- If a primary key has more than 1 column, when being used as a foreign key, all those columns will be included in the table which it is referencing.
- NOT NULL -> The column must have a value.
- A Foreign Key value can change, if the row (entity) to which it refers changes. Let us take an example of a 1:1 relationship where a user can own 1 car. So, there's a **Users** table and a **Cars** table. The former has **car_id** as FK and the latter has **username** as FK. So, **Users**(username, car_id, firstname, lastname, age) and **Cars**(car_id, username, car_name). Here, if the user changes his car, then his car_id will change.
- An FK shouldn't be NOT NULL. This is because, sometimes a relationship between one table to another isn't necessary. Eg - If a user has no car, then the car_id column in Users table will have to be NULL. If the FK was set to NOT NULL, then we couldn't accept users who didn't have cars.

- Another example would be an **Instructor** who teaches ***many*** **Classes**. Here, the INS_ID is the FK in the **Classes** table. If it has a NOT NULL condition, then we can't have a class without a teacher.
- Another example is of a relationship between a **Persons** table and a **Cards** table. Here, if we want to store all cards in the table (including those that have now owner and those that have expired), then we need to put a NOT NULL constraint on the FK person_id in the Cards table. Otherwise, if we want to store only valid cards that belong to different persons, we need not put that constraint.

Foreign Key Constraints (For MySQL)

- **FK Constraints** refer to the parent.
- **ON UPDATE** means that when we update the parent, we want the children to do something. **ON DELETE** means that when we delete the parent, then we want the children to do something.
- **RESTRICT**, **CASCADE** and **SET NULL** refer to what happens to the child tables.
- **ON DELETE RESTRICT** prevents the deletion of a row in the parent, if the FK of 1 or more children references that row in the parent.
- **ON UPDATE RESTRICT** prevents the updation of a row in the parent, if the FK of 1 or more children references that row in the parent.
- **CASCADE** will, like a domino effect, replicate whatever action done on the parent to all its children. Eg - If we delete a row in a parent with a certain PK value, then **ON DELETE CASCADE** will delete all rows with that particular FK value in all the children. **ON UPDATE CASCADE** cascades updates from the parent instead of deletes.
- The **ON DELETE SET NULL** makes the particular FK value in all children of a parent **NULL** if the row with the corresponding PK value in the parent is deleted.
- The **ON UPDATE SET NULL** makes the particular FK value in all children **NULL** if the row with the corresponding PK value in the parent has that PK value updated. When using these options, the FK in all the children shouldn't have the **NOT NULL** constraint set.

Simple Keys, Composite Keys and **Compound Keys**

- These terms are used for design purposes.
- A **Simple Key** consists of a single column.
- A **Composite Key** is a key that consists of multiple columns.
- A **Compound Key** is a key that consists of multiple columns in which each column is a key. Eg - The intermediary table studclass in the M:N relationship between students and classes. It would have a **compound key** consisting of **StudentID** and **ClassID,** each of which are keys themselves.

Review of Key Points

- A **Super Key** is any number of columns that forces every row to be unique.
- Candidate Key: The smallest Superkey.
- Primary Key: Any CK.
- Alternate Key: The other unchosen CKs.
- Surrogate and Natural keys.

Intro to Entity-Relationship Diagrams

- The entities are tables and are represented as boxes with their respective names.
- We define the row names in the entities.
- We define the type of relationship between the entities. One way to do it is to use Crow's Foot notation.

Cardinality

- There are 2 kinds of relationships possible in an ERD: 1:1 and 1:N. This is because M:N relationships are converted into two 1:N relationships joined together by an intermediary table.

Modality

- Modality tells us if a particular relationship between 2 entities is necessary or not.
- An example would be the 4 possible types of relationships between the **Cards** table & the **Cardholders** table.
- A **Cardholder** can be related to 0 or 1 **Card**.
- A **Cardholder** must be related to exactly 1 **Card**.
- A **Cardholder** can be related to 0 or N **Cards**.
- A **Cardholder** can be related to 1 or N **Cards**.
- M:N relationship isn't considered here due to the logic of the situation (One card can't belong to many individuals and vice-versa).
- The **NOT NULL** condition applies to any relationship where the modality allows 0.

Introduction to Database Normalization

- Normalization is a process where we check for and correct DB anomalies and eliminate repeating data.
- The successive Normal Forms are followed step by step like a plan to remove database anomalies and repeating data from the database.
- We have to ensure that everything is atomic.
- We also have to think about data depending on other data.

- **Dependence** -> An example of dependence is the example where the **age** of a user depends on the **username** (PK). This means that **abc123 -> 28** and **abc124 -> 29**. Here, **28** belongs to 1 entity and **29** belongs to another.
- The Normal Forms must be done in order. This means that we do 1NF, followed by 2NF followed by 3NF. It also means that a DB in 2NF is already in 1NF and a DB in 3NF is already in 1NF and 2NF.

First Normal Form (1NF)

- This Normal Form deals with Atomicity. It aims to ensure that all values stored in the database
- Eg - A multi-valued attribute like address needs to be broken down into single-valued attributes like street, house_number, state, country, postcode, etc. (For the table that contains the address) (This table has **user_id**, **fn**, **ln**, **email**, **address**).
- Eg - Each attribute in a row must contain only a single value. (For the table that contains the email column with 2 values in the first row). (Here, the values are the problem). (This table has **user_id**, **fn**, **ln**, **email**)
- Each row should be unique (For the example with 2 identical rows). (Here, the rows are the problem). (This table has **user_id**, **fn**, **ln**, **email**).
- So, since there are multiple emails for a user, the solution is to create an **Emails** table and have a 1:N relationship between **Users** and **Emails**.

Second Normal Form (2NF)

- It depends on a concept called **Partial Dependency**. **It aims to remove all Partial Dependencies from all the tables in the database.** This is a phenomenon that occurs only when a column depends on some parts of the PK. This occurs when the PK is a Compound Key or a Composite Key (i.e., it has multiple columns).
- A **Dependency** occurs when an attribute depends on the Primary Key. Eg - If we have 2 tables, **Users** and **Cars**, then all the users' attributes will be dependent on **user_id** while all the cars' attributes will be dependent on **car_id**.

- Let us take an example of an M:N relationship between a book and an author. 1 author can write many books and 1 book can be written by many authors. So, in this case, we have to have 3 tables: **Authors** (author_id, first_name, last_name); **Authors_Books** (author_id, book_id, author_position) and **Books** (book_id, isbn, book_name).
- The PKs in the 3 tables are **(author_id) in Authors**; **(author_id, book_id) in Authors_Books** and (**book_id)** in **Books**.
- Now, the author_position is fully dependent on the PK (author_id, book_id) in **Authors_Books**.
- If we were to put it in either **Authors** (which book ?) or **Books** (which author ?), it wouldn't make sense and it would be partially dependent on the PK of either table in case of being placed in either table.
- Another thing to consider is, if we put ISBN in **Authors_Books**, then it will be partially dependent on book_id and not the full PK (author_id, book_id).
- So, we have to ensure that **no partial dependencies exist after converting to 2NF**.

Third Normal Form (3NF)

- In this process, we remove transitive dependencies.
- Transitive dependencies are those of the form X -> Y -> Z.
- So, here we take the Y -> Z part of the dependency and put it in a new table with a new PK and add the PK of this new table as an FK in the table in which the transitive dependency existed.
- An example would be if we had a table **Reviews** (review_id, review_text, star, star_meaning). Here, the transitive dependency is **review_id -> star -> star_meaning**.
- So, what we do is we create a new table **Stars** (star_id, star, star_meaning) and we put **star_id** as an FK in **Reviews**.