DDL - Data Definition Language

DDL

Data Definition Language statements are utilized to define the database schema or skeleton. It is how we implement the design structure of it. Some of the keywords in this sublanguage are:

- CREATE, to create new objects or tables.

```
CREATE TABLE TABLE_NAME (C_NAME C_TYPE C_SYZE [NULL | NOT NULL], [CONSTRAINT])
```

- ALTER, to modify existing objects or tables.

```
ALTER TABLE TABLE_NAME [ADD | MODIFY | DROP] C_NAME
ALTER USER IDENTIFIED BY PASSWORD
```

- DROP, to delete existing objects or tables.

```
DROP TABLE TABLE_NAME [CASCADE]
```

- TRUNCATE, to delete all the data existing within a table leaving the skeleton of the table only.

```
TRUNCATE TABLE_NAME
```

This is similar to performing DELETE TABLE with nowhere clause, the key difference is that TRUNCATE commits at the end of the operation.

All DDL operations **cannot be rolled back**, which means that any change made by these are permanent.

DML - Data Manipulation Language

DML

Data Manipulation Language statements are used to perform CRUD operations on the actual data. Operations are normally performed by row in a relational database.

- INSERT, to insert a new row into a table.

```
INSERT INTO TABLE_NAME VALUES (V1, V2, ..., VN)
```

```
INSERT INTO TABLE_NAME (C1, C2, C3) VALUES (V1, V2, V3)
```

- UPDATE, to update one or more rows column values of a table that match a specific WHERE clause.

```
UPDATE TABLE_NAME SET C1 = V1, ... , CN = VN WHERE X = Y
```

- DELETE, to delete one or more rows of a table that match a specific WHERE clause.

```
DELETE TABLE_NAME WHERE [condition]
```

- SELECT, to obtain one or more rows of a table that match a specific WHERE clause. In ORACLE databases this one is considered DML. This is how we perform queries in a database.

```
SELECT C1, ..., CN FROM TABLE_NAME [table] WHERE [condition] GROUP BY [expression]
HAVING [condition] ORDER BY table.field
```

DQL - Data Query Language

DQL

Data Query Language, not really a sub-language within Oracle databases, is the sub language where only the SELECT statement exists. Remember, in Oracle the SELECT statement is considered as DML, but DQL could be an interview question.

DQL Clauses

GROUP BY & HAVING

The GROUP BY clause will combine all rows by a column specified in a query and perform any aggregate functions which are stated.

- `SELECT NAME, COUNT(NAME) FROM STUDENT GROUP BY (NAME)`

The HAVING clause will pass another filter similar to the WHERE clause after everything has been filtered and grouped.

- `SELECT NAME, COUNT(NAME) FROM STUDENT GROUP BY (NAME) HAVING COUNT(NAME) > 5;`

If you try to perform this HAVING clause in a WHERE clause, a SQL error will be thrown, and it makes sense - the RDBMS doesn't want you to perform an aggregate function combining all rows, per each row. It's a performance safety measure.

Scalar Functions

Scalar functions operate on individual values and will perform some operation per row, and can be used in the SELECT or WHERE clause.

- `TO_CHAR(DATE,'DATE_FORMAT')`
- `TO_DATE(DATE,'DATE_FORMAT')`
- `UPPER('VALUE')`
- `LOWER('VALUE')`

To write them in a query:

```
SELECT UPPER(NAME) FROM STUDENT;

SELECT NAME FROM STUDENT WHERE UPPER(NAME) LIKE 'P%'.
```

Aggregate Functions

Aggregate functions operate on multiple values (multiple rows). These functions are used to combine (aggregate) the values existing in one column.

- MAX(COLUMN), which returns the max value on a column.
- MIN(COLUMN), which returns the minimum value on a column.
- AVG(COLUMN), which returns the average value of the column.
- SUM(COLUMN), which returns the sum of the column.
- COUNT(COLUMN), which returns the count of elements in a column.
- And many others.

These functions are used in the SELECT clause. They can't be used in the WHERE clause.

If there is more than one column being selected in the SELECT column section of a query which is not aggregating, a GROUP BY clause is required.

In order to perform similar WHERE clause Boolean operations with aggregate functions, the HAVING clause can be used.

DCL - Data Control Language

DCL

Data Control Language statements are used to manage the security and control of database systems.

- GRANT, to grant any permissions to an existing user.

```
GRANT PERMISSION TO USERNAME
```

- REVOKE, to revoke any permissions of an existing user.

```
REVOKE PERMISSION TO USERNAME
```

Constraints

We can put integrity **constraints** on specific columns in our database when defining tables, which allow us to enforce the schema by ensuring consistency and integrity of the data in the table. The different constraints are listed below:

- PRIMARY KEY
- FOREIGN KEY
- NOT NULL
- UNIQUE
- CHECK
- DEFAULT
- AUTO INCREMENT

A **primary key** is a constraint that uniquely identifies a record in a table. Often, this constraint will be enforced on some sort of "ID" field, such as "employee_id". A primary key is inherently composed of two other constraints - unique and not null. Thus, a primary key *MUST* be provided when inserting a record into a table, unless the RDBMS system is generating it automatically behind the scenes.

A **foreign key** constraint signifies that a column represents a reference to the primary key of another table. This allows us to create relationships between tables. For example, if we are modeling cars and the owners of those cars, we might have a `Car` table with an `owner_id` foreign key that references the `user_id` field in the `People` table. We can then lookup the owner of any car by fetching the `owner_id` of the car and finding the matching `user_id` in the `People` table.

A **not null** constraint simply enforces that all records must have a field for the column on which this constraint is applied. For example, we know that every person has a social security number, so we might want to consider placing a not null constraint on that field in our users table (assuming we want to store the social security numbers). This prevents users of the database from leaving the table in an inconsistent or invalid state.
The **unique** constraint works similarly - records cannot be inserted if another record already has the same value for the column on which this is declared.

The **check** constraint provides a way of performing validation on values before records are entered into the table. For example, we may want to ensure that a bank account can never have a negative balance, so we might set a check constraint (`CHECK (balance >= 0)`).

Finally, a **default** constraint allows setting default values on columns for records that are inserted into the table.

Finally **Auto-increment** allows a unique number to be generated automatically when a new record is inserted into a table. Very often the *primary key* of a table needs to be created automatically, and we define that field as AUTO INCREMENT field. Following is the syntax for creating an AUTO INCREMENT field.

```
CREATE TABLE TableName (

Column1 DataType AUTO_INCREMENT PRIMARY KEY,

Column2 DataType,

);
```

Candidate and Composite Keys

Usually a primary key consists of a single column; however, sometimes we may have a scenario in which there could be multiple columns that together create a primary key to uniquely identify rows. We call these **candidate keys**. Once we identify the actual combination of columns to use as the primary key, we call this a **composite key**. For example, if you are modeling your CD collection, you might have fields such as `track_no`, `album_id`, and `genre` in the `Track` table. The `track_no` itself cannot work as a primary key because many different albums can have a track #1, for example. So we would need to create a composite key here, consisting of the `track_no` and the `album_id` columns. Using both of these columns together we can find the specific track we are looking for.

Referential Integrity

When we create table relationships as demonstrated above, it is important that our data remains in a consistent state throughout the database. For example, we never want a record on our class table to be pointing to a record in either the Teacher or the Student table that does not exist. We call enforcing this property as maintaining **referential integrity**. When we break referential integrity, we will find **orphan records** in the database - these are records whose foreign keys do not point to an existing record in the other table. One way of preventing this from occurring is by using a setting called **CASCADE DELETE** - when we enable this, deleting a record in the table will also cascade that operation and delete any records in tables that reference the record via foreign keys.

Table relationships / multiplicity

Multiplicity

As mentioned before, table relationships can be defined using foreign key constraints. There are several different kinds of relationships that exist between tables in relational databases:

- One to one
- One to Many / Many to One
- Many to Many

A one-to-one relationship means that each entity in the table only relates to a single entity in the other table. For example, if we are modeling a school, where each classroom has a single projector in it, we would want to make this relationship a one to one between the `Classroom` and the `Projector` tables. In our database, we can provide the classroom table a `projector_id` foreign key and provide the projector table a `classroom_id` foreign key. To enforce the one to one aspect, we should also apply a *unique* constraint on the foreign key columns. Otherwise, a user could add another projector record with the same `classroom_id` as an existing record, and then our one to one relationship would be broken.

A one to many (or vice versa, many to one) relationship is where one entity can belong to, own, or otherwise relate to multiple other entities. In our school modeling example, a Student could have many books, so this would be a one to many relationship. To create this in the database, we add the foreign key **only on the many side of the relationship** - so a book entity would have a field such as `student_id` as a foreign key to identify the owning student.

A many-to-many relationship implies a one-to-many relationship in both directions on the entities. For example, a Teacher can have many Students, but a Student could have many Teachers as well. In this case, **we cannot provide a direct link between the tables in the database - instead, we need to create what is called a *junction table* or *bridge table* to relate the two tables**. So, in our student-teacher example, we could create a `Class` table which contains two foreign keys - one that refers to the Teacher table's primary key and one that refers to the Student table's primary key. This creates a list of unique Teacher-Student mappings that can be used to look up which students a particular teacher teaches, or which teachers a particular student has. An example is shown below.

Class Table

ClassId	TeacherId	StudentId
1	1	1
1	1	2
2	1	3
3	2	1
3	2	3

We can see above that Teacher 1 teaches both Student 1 and 2 in the same class. Teacher 2 teaches Student 1 and 3 in a different class. Teacher 1 also has another class where he just teaches Student 3.

Normalization

Normalization refers to an optimization process of structuring a relational database in a way that *reduces redundancy* of data and improves data integrity and consistency. There are many different normal forms, which relate to the degree to which a database has been normalized. We will look at the first three normal forms, each of which build upon the previous:

- 1NF - must have a primary key, no repeating groups, and atomic columns
- 2NF - must already be in 1NF, plus have no partial dependencies
- 3NF - must already be in 2NF, plus have no transitive dependencies

The first normal form enforces that a table **must**:

- Have a primary key
- Each column should be as granular as possible (e.g. "Name" column should be broken up into: "First Name", "Last Name", "Middle Name", etc..)

To be in second normal form, a table must **also**:

- Cannot have columns that are dependent on only one part of the key
- If there are no composite primary keys, you are automatically in 2NF

Finally, to get to third normal form, a table must **also**:

- Not have transitive dependencies
- This means that if column C relates to column B which relates to column A which is the primary key, this is not in 3NF because C is related to the primary key but indirectly (it is a transitive dependency)

To advance into higher normal forms, we typically "break up" tables into multiple tables and relate them to each other via foreign keys.

A good way of remembering these normal forms in order is to remember the legal proceeding of swearing to tell the truth, the whole truth, and nothing but the truth. In relational databases, we must have **the key (1NF), the whole key (2NF), and nothing but the key (3NF)**.