DBMS Lecture 3

SQL Data Types:

1. Integers:
    - There are again different types of Integer data type that exist based on their ==size.==
        - TINY INT: 1 Byte (-128 to 127), (0 to 255).
        - (0 to 255) is called UNSIGNED TINY INT (more like a char/byte)
        - SMALL INT: 2 Bytes (-32k to 32k) (Doesn't really exist)
        - MEDIUM INT: 3 Bytes (-8M to 8M)
        - INT: 4 Bytes (-2B to 2B) (normal INT)
        - BIG INT: 8Bytes (-9Z to 9Z) (normal long long INT)

---

2. Floating Points:
    1. Decimal(p,s)
        - p: Total length/precision (1 to 64)
        - What can be the largest power of 10 we can store here??
        - There are 65 digits, so maximum power of 10 is 64, Now we can say decimal store more than floats, since float has max of 10^38^
            - s: How many values after decimal that we want to store.
    - Example: 302.04, p: 5, s:2
    - This p and s helps MYSQL store data like, Let's say internally MYSQL stores like this, 30204 then also MYSQL will be able to know exact decimal point.
    - The beauty of this is, it allows MYSQL to store exact values.

Float/Double: We often heard that Don't do comparisons with float/double because internally floating point numbers are not stored as it is.

For Example: `1.9-0.9==1` , ans: True But it might not be true, because internally they are like approximation, they use **IEEE** representation of floating point numbers (https://en.wikipedia.org/wiki/IEEE_754)check this link for details. So this is like an approximation not exactly.

Closest approximation of: `1.9=1.89, 0.9=0.91` programming language, ans:false. So this is not correct.

Let's say we are on a stock application, we can buy partial stocks. So let's say we buy a stock of coin 0.2, another person buys 0.01, another buys 0.5, another 0.6 and internally all of them are being stored as floats.

So let's say they are stored like: 0.2= 0.20001 0.01= 0.000009 0.5= 0.501001

Even though each of these numbers are still very close but now imagine the scale of a stock in an application. let's say it has almost 10^6^ orders a day, combining all these values, can the difference become huge? Even if there was a difference of 0.001 per order but combining it by 10^6^ that is the difference of 10^3^.

So one shouldn't store it in float/double because of the approximation that happens in floating point numbers. Even though each small number is still very close, but combining all of those numbers, the difference can become huge and it might leads to very random results.

For Example: Let's say if someone have to finally show the balance to be equal to 0, the balance will never come out to be 0 if too many floating numbers are used.

The reason for using floating point numbers are: 1. Huge range of numbers they can store. 2. They store scientific notations. 3. Used in scenerioes where approximation is ok. 4. Sometimes we write extra logic to handle approximation.

---

2. Float: - 4 Byte number. - (-3.4* 10^38^ to 3.4* 10^34^)
3. Double: - 8 Bytes - (-1.7* 10^308^ to 1.7* 10^308^)

- Both of them is going to store approximation. - If we need to store exact values, don't use them or have to use extra logic to handle approximation.

---

3. BOOLEAN: (BOOL or BOOLEAN)
    - Internally a boolean is nothing but a Tiny int.
    - Size of boolean: 1Byte
    - 2 values: True or False/ 1 or 0

---

4. ENUMS:
    - A set of constants.
    - Benefit of Enum: With enum compiler is going to ensure that we don't end up inserting something that is not a valid value.
    - Value should be amongst the given set of values.
    - For Example: In a cars table, there is a car_type column and we specify them using enum. so enum("BMW", "sedan", "audi") and if we try to put any value other than this, it will throw error.
    - Problem: In every car_type row, a string is going to be stored, What if we decide to change the name of car type.
    - For Example: "Audi" will be called "Oudi", We will have to update all the rows, this is slow.
    - Practical advice: Don't have enum columns.
    - Have a seperate table.

---

5. Date and Time: (A seperate class is going to be there on built in SQL functions).

- Date: - only going to allow to store a date. - Like, 28/3/2022 - Time: - Only allow to store the time part. - Like, 6:00pm - DateTime: - Allows to store both date and time. - Like, 28/3/2022 6:00pm

---

TIMESTAMP: Miliseconds since 01/01/1970 UTC. https://www.epochconverter.com(Check this link)

- This basically eases down a lot of internationalization problem.
- Don't do this in MYSQL.
- In MYSQL, Timestamp is 4 bytes.
- In 2038, this will exhaust. 01/01/2038
- Because the number of seconds since 1970> range of timestamp. This lead to: All of the databases that are using timestamp as the column to store a particular time are going to ==overflow.==
- https://en.wikipedia.org/wiki/Year_2038_problem(Read more from this).

BLOBS: - Binary Large Objects. - Storing files in database. TINY BLOB: 255B NORMAL BLOB: 65KB MEDIUM BLOB: 16MB LONG BLOB: 4GB - Don't use these unless explicit reason. - Often a file store is going to be a better choice. - Technically, Facebook don't store all the photos of a user in database, can do that but actually don't do it. - Store metadata of file in database and store file in a file store(AWS S3) - Metadata of file: fileURL, fileId.

Normalization:

- A techinque to reduce ==redundancy==(of data) in a database.
- Helps to create better database schema(Schema with reduced redundancy).
- Works by mostly creating extra tables to reduce redundancy.

Problems (Anomalies) with redundancy:

Let's take the example of scaler: We are trying to create a database for scaler. In the database, we have a table called students.

Now, If this is the database table, there will be some redundancy when we put the data in the table.

What we did here, Instead of creating seperate tables for students and batches, We merged both tables in one, so this students table have information about batches as well as students.

But the problem is Redundancy with this database.

So, student who belong to batch 1 let's say, there name is A, instructor is B, so wherever there is batch_id 1, A and B will be repeated. So there is redundancy, the information about batch_name and batch_inst is redundant.

Different problems due to Redundancy:

1. Insertion Anomaly: - We can't create a batch till no student in there. since, Id can't be NULL, Even then nothing is there in the students. - This is the problem.

2. Deletion Anomaly: Let's say, Scaler decided that all the students of batch A and C should go to batch D, instructor B and id 3. Now, the problem is there is not a single information left about A, C anymore. A and C are completely gone from the system. Even the system has merged but A has other information also, the assignments, classes, previous instructors. That is losing information, so this is not a good idea. - Deleting other entities might end up deleting batch as well.

3. Update Anomaly: Now, Scaler wants to change the instructor of batch A to G. It'll write an update query.

Let's say after this some error happened, electricity fault, or some corruption. Now we are at this state, ==Inconsistent data.==

So we have to avoid these problems, reduce redundancy, make the database design better. For which we use normalization.

DB Normalization:

- Process to organize the data in a a database.
- Reducing redundancy is the biggest motive of database normalization.
- There are multiple normal forms, each form is stricter than the previous one.

Normal Forms:

1NF:

- Every column should have atomic value.
- No multi values attributes.
- Example:

Solution1: Problem with this design:

Now, Let's say person1 only have 2 ph_no., person2 only 1, We are having too many ==NULL, sparse table.==

Solution2: Seperate table.

If we have the id column, the primary key of phone-number table is Id.

If we don't have this, then st_id and ph_no combined will be the primary key. **Benefit of this:** It is sorted on st_id, that means if we want all the ph_no of a particular student, it's going to be faster.

2NF:

Let's take a scenerio. Scaler is storing a database table called mentor_sessions. mentor_sessions table is responsible about information related to mentor sessions. Now, If we have mentor_id, We don't really need mentor_name here.

Primary key: First 3 columns because 1 student with 1 mentor can have mutiple sessions.

Now, we can say the attribute mentor_name can just be found via mentor_id. This mentor_name is actually not depending on all the 3 columns but just mentor_id, a part of primary key.

==According to 2NF, No non-prime attribute(attribute not part of primary key) should depend on subset of primary key.==

Here, mentor_name is dependent upon a subset of primary key. This is not allowed.

- All the non prime attributes should depend on complete primary key.

3NF:

- No non prime attribute should indirectly depend on primary key. Let's say we have students table, Primary key: id. name completely dependent on id, batch_id is dependent on id, batch_name is also dependent on id. But from student id, we will get batch_id and from batch_id we will get their batch_name. 2NF is followed.

But batch_name is not directly dependent on complete primary key it is transitively dependent. Not 3NF.

`You can use these normal forms to evaluate the already existing schema but not really to create the new schema.`

In practical, Denormalization has benefits as well. If we see that the database is not following the 2NF, We will Refactor it, by creating new tables.

Now, If we create more and more tables, When we will be doing queries, we will do lot of joins and joins need a lot of time of CPU, It's expensive. So that's why when we have a large scale database or when we need faster queries, often we denormalize the data, We don't need this normalization because that is slowing the queries.

SQL:

CRUD:

Whenever an entity is stored, clients typically do CRUD operations on those entities. CRUD is an acronym for 4 terms: C: Create R: Read U: Update D: Delete

https://drive.google.com/file/d/1ftbQW_Tud46ihfnh2BTlBnw6NQuOHpKp/view?usp=sharing Use this SQL script as a pre text. Download this SQL script and then Go to file in MYSQL Workbench, Open SQL script, Download and then click on Run button. It's a sample data.

Create Data:

INSERT:

==SQL is case insensitive. INSERT=insert=INseRt==

```
insert into [table_name] values
(           )
```

Go to sql_store then in customers table. Let's insert a new customer.

```
insert into sql_store.customers
values (11, naman, bhalla, '1998-01-02, 123', 'ABC', 'XYZ', 'HR',1134),
(12, naman, bhalla, '1998-01-02, 123', 'ABC', 'XYZ', 'HR',1134);
```

Run this (command+enter) or run query button. There are 2 new customers inserted in database.

Go to table defination, Id is auto_increment. Id is a primary key, auto_increment is we can generate the primary key ourself whatever is tha past key that we had just add 1 to it and that is going to be the new primary key. So now database will automatically generate the value.

```
insert into sql_store.customers
values (default, naman, bhalla, '1998-01-02, 123', 'ABC', 'XYZ', 'HR',1134),
(default, naman, bhalla, '1998-01-02, 123', 'ABC', 'XYZ', 'HR',1134);
```

Another variant:

```
insert into sql_store.customers (customer_id, first_name,last_name,birth_date,phone,address,city,state
values(default, naman, bhalla, '1998-01-02, 123', 'ABC', 'XYZ', 'HR',1134),
(default, naman, bhalla, '1998-01-02, 123', 'ABC', 'XYZ', 'HR',1134);
```

It is making the query a bit more understandable.

Now, there are some columns that are nullable also, birth_date,phone, so we can remove that.

```
insert into sql_store.customers (first_name,last_name,address,city,state,points)
values(naman, bhalla, 'ABC', 'XYZ', 'HR',1134),
(naman, bhalla, 'ABC', 'XYZ', 'HR',1134);
```

So, It has put the values, only difference is the values which we didn't provide have NULL.

- We don't need to provide values to columns that are(In second variant when we give the column name)
- Nullable
- Default value.

Another benefit of giving the column names is: These columns can be in any order.

```
insert into sql_store.customers (first_name,last_name,address,city,state,birth_date,points)
values(naman, bhalla, 'ABC', 'XYZ', 'HR','1998-01-02',1134),
(naman, bhalla, 'ABC', 'XYZ', 'HR','1998-01-02',1134);
```

This means that if we specify the column name ourself, the column names need not to be in the same order as in table.

Update: - We want to change the value of something that is stored.

Update [table_name] set {give values} condition(optional)

Example in scaler codebase: `update students set name='naman'` No condition: Therefore names of all the students become naman.

Now, We want to update the name of customer_id 21 to manish yadav.

```
Update sql_store.customers set{first_name='manish', last_name='yadav' where customer_id=21};
```
