COURSEWARE

Data Definition Language (DDL)

Contents

Overview

Data Definition Language (DDL) is one of the subsets of SQL query types.

It centers around defining the schema for the database, and writing queries to create, alter and delete parts of the schema.

Creating a Database

The SQL statement to create a database is as follows:

```
CREATE DATABASE some_database;
```

You can also use the optional `IF NOT EXISTS` command to ensure that your database will only be created if it does not already exist:

```
CREATE DATABASE IF NOT EXISTS some_database;
```

From here, if you wish to work with this database, you would run the below:

```
USE some_database;
```

Creating Tables

DDL-based SQL statements can be used to create tables.

The syntax for this is similar to the below:

```
CREATE TABLE table_name (
  column_name1 data_type(size) constraint_name,
  column_name2 data_type(size) constraint_name,
  PRIMARY KEY (field_name),
  FOREIGN KEY (other_field) REFERENCES table_name(primary_key)
);
```

- `CREATE TABLE` is used first alongside the name of the table, followed by an open parenthesis to declare table fields
- We then list the fields in order of field name, data type (with the size in parentheses if relevant) and a constraint we wish to use

- We can create a primary key with the `PRIMARY KEY` keyword, with the PK field in parentheses
- We can also create a `FOREIGN KEY`, with the first set of parentheses containing the field in the current table we wish to hold a reference, and the second set of parentheses specifying the field which we are referencing in a different table.
- Make sure you put a comma between different columns and a semi-colon at the end of your command.

Constraints

We can add constraints to our data to further improve data validation beyond data types for fields.

Data types is a simple method of validation, but we can make this more complex with constraints, such as:

- `NOT NULL`
- `UNIQUE`
- `DEFAULT`
- `PRIMARY KEY` and `FOREIGN KEY`

NOT NULL

`NOT NULL` is a constraint that ensures a column cannot have a `NULL` value - the absence of data. In other words, it cannot be empty:

```
CREATE TABLE my_table (
  name varchar(100) NOT NULL
);
```

It is important to bear in mind the difference between a *blank* field and a `NULL` one:

- `NULL` fields are entirely empty of any data whatsoever
- *Blank* fields may contain some kind of data, whether it is hidden or a single space character (" "), but it is *not* completely empty

If you insert spaces into a field with the `NOT NULL` constraint, that is compliant as technically it is *not* empty.

UNIQUE

The `UNIQUE` constraint ensures that each column must have a unique value when compared to other records with values in that field.

```
CREATE TABLE my_table (
  name varchar(100) NOT NULL,
  rank int UNIQUE NOT NULL
);
```

DEFAULT

We can also have a `DEFAULT` value for columns, which will auto-fill when we create records.

If no value is specified during data entry, this is the value that will be used:

```
CREATE TABLE my_table (
  name varchar(100) NOT NULL,
  rank int UNIQUE NOT NULL,
  status varchar(10) DEFAULT 'unranked'
);
```

AUTO_INCREMENT

The `AUTO_INCREMENT` command assigns a whole-number numeric field (INT, BIGINT etc.) the ability to automatically increment its value when a new record is inserted.

By default, the starting value is `1`:

```
CREATE TABLE my_table (
  person_id int AUTO_INCREMENT,
  name varchar(100) NOT NULL,
  rank int UNIQUE NOT NULL,
  status varchar(10) DEFAULT 'unranked'
);
```

If we wanted to insert a new record into our table, we would not need to indicate that we wanted the `person_id` column filled.

PRIMARY KEY

A `PRIMARY KEY` is a combination of the NOT NULL and UNIQUE constraints, and allow us to uniquely identify records from one another:

```
CREATE TABLE my_table (
  person_id int AUTO_INCREMENT,
  name varchar(100) NOT NULL,
  rank int UNIQUE NOT NULL,
  status varchar(10) DEFAULT 'unranked',
  PRIMARY KEY(person_id)
);
```

A table can only have one `PRIMARY KEY` - and, usually, a table will contain one.

A `PRIMARY KEY` can be made up of two fields, also known as a *composite key*;

```
CREATE TABLE example (
    field1 INT,
    field2 INT,
    PRIMARY KEY (field1, field2)
);
```

Imagine you had a `customers` and a `products` table like so:

`Customers`

id	name	email
1	Bill S. Preston, Esq.	bill.preston@excellent.com
2	Ted "Theodore" Logan	theodore.logan@partyon.com

`Products`

id	product	price
1	Les Paul	3000.00
2	Evil Robot Dude	7319.99
3	Time Machine	1000000.69

And an `orders` table with foreign keys referring to the IDs for customers and products:

`Orders`

fk_customer_id	fk_product_id	quantity	total	date_placed
1	1	1	3000.00	12/3/89 13:53:32

fk_customer_id	fk_product_id	quantity	total	date_placed
2	1	2	6000.00	12/3/89 14:04:13
1	3	1	1000000.69	03/6/89 20:21:58

None of the columns in the `orders` table can be used as primary keys by themselves because none of them will be unique: a customer can make multiple orders, the same product can be ordered multiple times and multiple orders can be placed at the same time.

Instead, we can create a composite key that combines `fk_customer_id`, `fk_product_id` and `date_placed` to create a unique identifier for each order placed. The same customer cannot order the exact same product at the exact same time, making this composite key unique.

The SQL syntax to create this table will therefore be:

```
CREATE TABLE orders (
      fk_customer_id INT NOT NULL,
      fk_product_id INT NOT NULL,
      quantity INT NOT NULL,
      total DEC(10,2) NOT NULL,
      date_placed DATETIME NOT NULL,
      PRIMARY KEY (fk_customer_id, fk_product_id, date_placed),
      FOREIGN KEY (fk_customer_id) REFERENCES customers(id),
      FOREIGN KEY (fk_product_id) REFERENCES products(id)
);
```

FOREIGN KEY

A `FOREIGN KEY` ensures that we can cross-reference data from one table to another, marking the link between tables and creating relationships.

A `FOREIGN KEY` field in one table will point to a `PRIMARY KEY` field in another table, which we can then use to refer to the rest of the data stored inside the record containing the `PRIMARY KEY`:

```
CREATE TABLE my_table (
  person_id int NOT NULL AUTO_INCREMENT,
  name varchar(100) NOT NULL,
  rank int UNIQUE NOT NULL,
  status varchar(10) DEFAULT 'unranked',
  PRIMARY KEY (person_id)
);
```

```
CREATE TABLE my_second_table (
  account_id int NOT NULL AUTO_INCREMENT,
  fk_person_id int NOT NULL,
  account_type varchar(100) NOT NULL,
  PRIMARY KEY (account_id),
  FOREIGN KEY (fk_person_id) REFERENCES my_table(person_id)
);
```

Altering tables

`ALTER TABLE` is used to add, delete, or modify fields in a table.

This becomes remarkably more difficult when data is present, so make sure the design of your database is as fool-proof as possible!

To add a field to a table:

```
ALTER TABLE table_name
ADD column_name data_type;
```

And to delete a field from a table:

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

Schema Deletion

Deleting a table can be done with the `DROP TABLE` statement.

You can list multiple tables if more than one needs to be dropped, and table data and definitions are all removed, so use with caution.

If not all the listed tables exist, MySQL will still drop all tables that do exist.

```
DROP TABLE IF EXISTS table_that_should_not_be_dropped;
```

Tutorial

There is no tutorial for this module.

Exercises

Using the tables you made in the `Data Design` module, now create the tables for your database using Data Definition Language.

You will need to create the database first and then swap to it:

```
CREATE DATABASE IF NOT EXISTS gameshopdb;
USE gameshopdb;
```

▶ Solution