Lecture 5: SQL's Data Definition Language

CS1106/CS6503– Introduction to Relational Databases

Dr Kieran T. Herley

2019-2020

School of Computer Science & Information Technology
University College Cork

Data Definition

Data Definition and Data Manipulation

- So far we've looked at SQL's *data manipulation language* (DML)
  - Extract information from table
  - Manipulate table by adding/removing rows of changing values
- How to we specify and create a table in the first place?
- SQL's also includes a *data definition language (DDL)* that allows us to
  - Specify the structure of your database's table(s)
  - Create a table with this structure

A Table Definition for Our Running Example

```
CREATE TABLE students
(
    id_number CHAR(9),
    first_name  VARCHAR(20),
    last_name VARCHAR(30),
    date_of_birth  DATE,
    hometown VARCHAR(30),
    course  CHAR(5),
    points  INTEGER,
    . . .
);
```

- Creates table named `students` with specified structure
- Structure
  - specified as list of attribute names and associated *types*
  - judgement required in choosing appropriate types (and lengths)
- (Health warning: Above could be improved!)

Overview of SQL's Data Types

- Remember that databases are used in a huge variety of applications to house very diverse types of data
- SQL provides a range of different "data types" with which to populate our tables

  Textual Strings of characters

  Numerical Numbers both integers and real

  Temporal Dates, times, date-time timestamps

  Others

  - Also other types: BLOB (Binary Large Object)
  - Most DBMSs support many flavours; often system dependent

Integer Types

- For integer (whole) numbers we use INTEGER (or INT) e.g.

  num_points **INTEGER**

- Range of allowed values (MySQL [1])

$$-2,147,483,468 \cdots 2,147,483,467$$

  MySQL should be good enough for most purposes.

- Most systems offer variants (SMALLINT, BIGINT) with different ranges; require different amount of space; need to choose carefully

[1] Limits typically DBMS specific.

Other Numerical Types

DECIMAL(n, d)

- any n-digit number with a d-digit mantissa
-

  temperature **DECIMAL**(4, 1)

  123.4 ok; 12.34 rounded to 12.3; 1234.5 error

FLOAT

- scientific notation (e.g. `1.34E+12` for $1.34 \times 10^{+12}$)
- Useful for scientific data
- system-dependent limits on number size

Temporal Types

DATE

- Dates in `YYYY-MM-DD` format

TIME

- Time (24-hour clock) in `hh:mm:ss` format

Others e.g. DATETIME[MySQL]

NOTES DBMSs support useful *functions* for manipulating dates and times e.g. `MONTH(.)`, `DATE_ADD(.,.)`[MySQL]

Textual Types

CHAR(n)

- Shortish, fixed-length strings
  - Space for exactly *n* characters allocated
  - shorter strings right-padded with blanks, longer ones truncated
- Useful for id numbers, course codes etc.

VARCHAR(n)

- Strings of any any length up to max of *n* characters
- May be more space-efficient than CHAR
- Useful where precise string length not fixed or known in advance, e.g. addresses

Note Both draw individual characters from underlying character set– `latin1_swedish_ci` by default in our case

Textual Types cont'd

- Need to be careful in choosing appropriate "size"
- *n* too large: wasteful of space (CHAR)
- *n* too small: String may be truncated if assigned to a CHAR(n)/VARCHAR(n) column of insufficient "size" i.e only first *n* characters retained

Meanwhile Back At The Example

- ```
  CREATE TABLE students
  (
      id_number CHAR(9),
      first_name  VARCHAR(20),
      last_name VARCHAR(30),
      date_of_birth  DATE,
      hometown VARCHAR(30),
      course  CHAR(5),
      points  INTEGER,
      . . .
  );
  ```
- Should only execute CREATE once when table is first set up
- Can also specify other useful information within CREATE statement: Information about DB keys, Default values for columns, Constraints on values. Will look at this later.
  ```

Other Structure-Altering SQL Commands

- To expunge a table:

```
DROP TABLE X;
DROP TABLE IF EXISTS Y;
```

  - Careful– deletes table and contents
- Altering table structure
  - Adding an attribute/column:

```
ALTER TABLE students ADD gender CHAR(1);
```

  - Deleting an attribute/column:

```
ALTER TABLE students DROP hometown;
```

    If you design your DB properly, you should rarely need these

Keys

students

id_number	*first_name*	*last_name*	*date_of_birth*	*hometown*	*course*	*points*
112345678	Aoife	Ahern	1993-01-25	Cork	ck401	500
112467389	Barry	Barry	1980-06-30	Tralee	ck402	450
112356489	Ciara	Callaghan	1993-03-14	Limerick	ck401	425
112986347	Declan	Duffy	1993-11-03	Cork	ck407	550
112561728	Eimear	Early	1993-07-18	Thurles	ck406	475
112836467	Fionn	Fitzgerald	1994-06-13	Bandon	ck405	485

- Each table should have one or more attributes (collectively known as the *key*) the values of which uniquely identify each row i.e. no two rows should have the same key
- Example: id_number plays this role in `students`
- Table definition should specify key as shown

```
CREATE TABLE students
(   . . .
    PRIMARY KEY (id_number)
)
```


- Yes, there are also non-primary keys; more on this later

Table Specification

```
CREATE TABLE guests
(   id_number INTEGER AUTO_INCREMENT, -- new rows automatically issued fresh id
                                      -- use with primary key column only
    name VARCHAR(20) NOT NULL,        -- inserting/setting NULL will trigger error
     nationality  VARCHAR(20) DEFAULT 'Irish'-- all guests Irish unless specified otehrwise

    PRIMARY KEY (id_number)
);
```

Simple Possible Refinements MySQL

Table Specification

```
CREATE TABLE guests
(   id_number INTEGER AUTO_INCREMENT, -- new rows automatically issued fresh id
                                      -- use with primary key column only
    name VARCHAR(20) NOT NULL,        -- inserting/setting NULL will trigger error
     nationality  VARCHAR(20) DEFAULT 'Irish'-- all guests Irish unless specified otehrwise

    PRIMARY KEY (id_number)
);
```

Some Insertions

```
INSERT INTO guests (name, nationality) VALUES
    ('Paddy', ' Irish '),    ('Gunther', 'German'),    ('Luigi', ' Italian ');
INSERT INTO guests (name) VALUES
    ('Seamus');
INSERT INTO guests (name, nationality) VALUES
    (NULL, 'Irish');                                  -- ERROR!!!
```

Creating and Populating A Database

- cs1106 website houses two files used to create clones of students DB in your accounts
  - `students_setup.sql`: file containing `CREATE` statement
  - `students_populate.sql`: file containing a bunch of `INSERT` statements
- MySQL can process SQL commands taken from a file not just those type in; in HeidiSQL use the folder icon to the right of the blue execution button
- We will provide files for most of the sample DBs we use in this module

An Example

A Simple Database

- Suppose we want to design a DB to hold information about some people
- Information about each person:
  - Name
  - Birth date
  - Address
  - Favourite foods

A Simple Database

- Suppose we want to design a DB to hold information about some people
-

  Information about each per-First stab at a DB design:
  son:

  - Name
  - Birth date
  - Address
  - Favourite foods

Column	Type
name	VARCHAR(?)
gender	CHAR(1)
birth_date	DATE
address	VARCHAR(?)
favourite_foods	?????

A Simple Database

- Suppose we want to design a DB to hold information about some people
- Information about each person:
  - Name
  - Birth date
  - Address
  - Favourite foods

First stab at a DB design:

Column	Type
name	VARCHAR(?)
gender	CHAR(1)
birth_date	DATE
address	VARCHAR(?)
favourite_foods	?????

- Some imperfections with this
  - Best to have unique label for each person in case of duplicate names; introduce ID numbers
  - Names are really composite objects; split into first and last names; same with addresses
  - Need to choose sensible values for VARCHAR widths

A Second Stab

Column	Type
person_id	CHAR(6)
first_name	VARCHAR(20)
last_name	VARCHAR(20)
gender	CHAR(1)
birth_date	DATE
street	VARCHAR(30)
town	VARCHAR(30)
county	VARCHAR(30)
favourite_foods	?????
- What to do about favourite foods?
  - VARCHAR– difficult to access individual food items
  - Separate columns (`fav1`, `fav2`, . .), but how many?
  - Better to use second separate table to capture this "relationship" (persons to food types)

The `favourite_foods` Table

- **persons**

person_id	*first_name*	*last_name*	. . .
112356489	Ciara	Callaghan	· · ·
112986347	Declan	Duffy	· · ·

favourite_foods

person_id	*food*
112356489	Ice cream
112356489	Chocolate
112986347	Pizza
112986347	Beer
112986347	Crisps

- This models the fact that person 112986347 (aka Declan Duffy) likes pizza, beer and crisps
- The "link" between the two tables is the `person_id`; within the `favourite_foods` table the `person_id` is a *foreign key* that references the `persons` table.

Our Complete Design

```
CREATE TABLE persons
(
    person_id    CHAR(6),
    first_name   VARCHAR(20),
    last_name    VARCHAR(20),
    gender       CHAR(1),
    birth_date   DATE,
    street       VARCHAR(30),
    town         VARCHAR(30),
    county       VARCHAR(30),

    PRIMARY KEY (person_id)
);
```

```
CREATE TABLE favourite_foods
(
    person_id    CHAR(6),
    food         VARCHAR(20),

    PRIMARY KEY (person_id, food)

);
```

Note: two-attribute key

MySQL enforces key-distinctness property for tables, e.g. disallow insertion of duplicate person_id values into `persons`

Working With Our New DB

- Can pose queries at either table using SQL tools we've learnt

```
SELECT *
FROM persons
WHERE first_name = 'Ciara';
```

```
SELECT *
FROM favourite_foods
WHERE person_id = '112986347';
```

- What about queries like the following?

List names of all persons who like pizza

What We Have Covered So Far

- Setting up a simple database (CREATE)
- Adding content to the database (INSERT, UPDATE)
- Posing (simple) queries to extract information from database

Notes and Acknowledgements

The favourite foods example is taken from “Learning SQL” by Alan Beaulieu (O’Reilly, 2009). If you are looking for a nice, compact and affordable introduction to SQL, this is a good choice.