```
/* ****************************************************************************
                    DATABASE ADMINISTRATION FUNDAMENTALS:
                  INTRODUCTION TO STRUCTURED QUERY LANGUAGE
                    SF21SQL1001, 2021/11/02 - 2021/12/09
                 https://folvera.commons.gc.cuny.edu/?cat=29
****************************************************************************

 SESSION #8 (2021/11/30): CREATING DATABASE OBJECTS

 1. Altering databases, schemata, tables
 2. Understanding `NULL` and `NOT NULL`
 1. Parameters, user-defined functions and stored procedures
****************************************************************************

1. The following set of concepts that is good for you to know involve how
   humans communicate with the computer and vice versa.

        ``A command line interface (CLI) is a text-based user interface (UI)
        used to view and manage computer files.  Command line interfaces are
        also called command-line user interfaces, console user interfaces and
        character user interfaces...
        Before the mouse, users interacted with an operating system (OS) or
        application with a keyboard. Users typed commands in the command line
        interface to run tasks on a computer.
        Typically, the command line interface features a black box with white
        text.  The user responds to a prompt in the command line interface by
        typing a command.  The output or response from the system can include
        a message, table, list, or some other confirmation of a system or
        application action.
        Today, most users prefer the graphical user interface (GUI) offered by
        operating systems such as Windows, Linux and macOS.  Most current
        Unix-based systems offer both a command line interface and a graphical
        user interface.
        The MS-DOS operating system and the command shell in the Windows
        operating system are examples of command line interfaces.  In
        addition, programming languages can support command line interfaces,
        such as Python.``
        https://searchwindowsserver.techtarget.com/definition/command-line-
         interface-CLI

        ``A GUI (usually pronounced GOO-ee) is a graphical (rather than purely
        textual) user interface to a computer.  As you read this, you are
        looking at the GUI or graphical user interface of your particular Web
        browser.  The term came into existence because the first interactive
        user interfaces to computers were not graphical; they were
        text-and-keyboard oriented and usually consisted of commands you had
        to remember and computer responses that were infamously brief.  The
        command interface of the DOS operating system (which you can still get
        to from your Windows operating system) is an example of the typical
        user-computer interface before GUIs arrived.  An intermediate step in
        user interfaces between the command line interface and the GUI was the
        non-graphical menu-based interface, which let you interact by using a
```

```
         mouse rather than by having to type in keyboard commands.
         https://searchwindevelopment.techtarget.com/definition/GUI

2. Now that we are going to start programability, we use parameters to pass
   values either to a SQL script and/or receiving parameters from external
   programs, built-in and/or user-defined procedures and/or functions.

         ``In information technology, a parameter (pronounced puh-RAA-meh-tuhr,
         from Greek for, roughly, through measure) is an item of information
         -- such as a name, a number, or a selected option -- that is passed
         to a program by a user or another program.  Parameters affect the
         operation of the program receiving them.``
         http://whatis.techtarget.com/definition/parameter

         ``Parameters can be passed to the stored procedures.  This makes the
         procedure dynamic.
         The following points are to be noted:
         * One or more number of parameters can be passed in a procedure.
         * The parameter name should proceed with an @ symbol.
         * The parameter names will be local to the procedure in which they are
           defined.
         The parameters are used to pass information into a procedure from the
         line that executes the parameter.  The parameters are given just after
         the name of the procedure on a command line.  Commas should separate
         the list of parameters.
         * The values can be passed to stored procedures by:
         * By supplying the parameter values exactly in the same order as given
           in the CREATE PROCEDURE statement.
         * By explicitly naming the parameters and assigning the appropriate
           value.``
         http://devguru.com/technologies/t-sql/7132

   Every time users pass values to a query commonly using a web form, you are
   at risk of SQL injections where the user could pass a SQL statement, which
   the server may execute.  For this reason, every database should have a
   read-only account that queries data returning values to the front-end
   application limiting the possibility of SQL injections and similar exploits
   (http://searchsecurity.techtarget.com/definition/exploit).

         ``SQL injection is a type of security exploit in which the attacker
         adds Structured Query Language (SQL) code to a Web form input box to
         gain access to resources or make changes to data.  An SQL query is a
         request for some action to be performed on a database.  Typically, on
         a Web form for user authentication, when a user enters their name and
         password into the text boxes provided for them, those values are
         inserted into a SELECT query.  If the values entered are found as
         expected, the user is allowed access; if they aren't found, access is
         denied.  However, most Web forms have no mechanisms in place to block
         input other than names and passwords.  Unless such precautions are
         taken, an attacker can use the input boxes to send their own request
         to the database, which could allow them to download the entire
         database or interact with it in other illicit ways.``
```

```
104           http://searchsoftwarequality.techtarget.com/definition/SQL-injection
105
106       2.1. In the example below, we first declare parameter `@foo` and `@pi`
107           (always starting with the `@` sign) to tell SQL Server that it should
108           expect a value within SQL script and/or receiving a value from an
109           external programs, built-in and/or user-defined procedure and/or
110           function.
111
112           Note that there are no limit to the number of parameters you can use
113           in a SQL script.
114
115           Go to https://en.wikipedia.org/wiki/Foobar if you are interested about
116           terms `foobar`, `foo` and `bar`.
117
118           We first declare parameters `@foo` with data type INT and `@pi` with
119           data type FLOAT.
120
121           We then set (assign) a value to parameters `@foo` and `@pi`.
122
123           We can then call the value of `@foo` and `@pi` within a SQL statement.
124   ************************************************************************** */
125
126  DECLARE @foo INT = 3,                              -- 1. declaring parameter
127                                                     --    `@foo` as INT initialized
128                                                     --    with value of 3
129    @pi FLOAT = 3.14159265358979;                    -- 2. declaring parameter `@pi`
130                                                     --    as FLOAT initialized with
131                                                     --    value of 3.14159265358979
132
133
134  /* ************************************************************************
135      2.1. At this point, we can call the values of parameters `@foo` and `@pi`
136           in a SQL statement.
137
138                     +-------+------------------+------------------+
139                     | foo   | pi(e)            | foo pi(e)        |
140                     +-------+------------------+------------------+
141                     | 3     | 3.14159265358979 | 9.42477796076937 |
142                     +-------+------------------+------------------+
143
144           Note that we use a `SELECT` statement in the same way we use `PRINT`
145           in other languages to show the output in the console.
146   ************************************************************************** */
147
148  SELECT @foo AS 'foo',                              -- 1. displaying value of
149                                                     --    `@foo` (INT) with alias
150                                                     --    `foo`
151    @pi AS 'pi(e)',                                  -- 2. displaying value of
152                                                     --    `@pi` (FLOAT) with alias
153                                                     --    `pi(e)`
154    @foo * @pi AS 'foo pi(e)';                       -- 3. displaying value of
155                                                     --    `@foo` multiplied by
```

```
                                                --    `@pi`, which returns a
                                                --    FLOAT due to data type
                                                --    conversion with alias
                                                --    `foo pi(e)`


/* ***************************************************************************
 3. ``In SQL Server, a procedure is a stored program that you can pass
    parameters into.  It does not return a value like a function does.
    However, it can return a success/failure status to the procedure that
    called it.``
    https://techonthenet.com/sql_server/procedures.php

                    CREATE PROCEDURE procedure_name [@input_param data_type]
                    AS
                    BEGIN
                      [DECLARE @output_param data_type
                      SET @output_param = some_value]
                      executable_code
                    END;

    This means that we can take the code that we used to capitalize the first
    letter in a string and make it into a procedure that we can call indicating
    the input parameter instead of writing the same code several times and
    avoid the possibility of errors.

                    SELECT CONCAT (
                      UPPER(LEFT(`hello`, 1)),
                      LOWER(SUBSTRING(`hello`, 2, LEN(`hello`) - 1))
                      );

    3.1. In the example below, we declare function `AP5.properUDP`.  We end
         the name of the function with `UDP` to identify it as an user-defined
         procedure.

         The procedure has input parameter `@in_string` declared as VARCHAR(50)
         -- in this case, the string `hello`.

         We enclose the executable section between `BEGIN` and `END`.

         We create output using parameter `@out_string`, which must have the
         same file type as the input, in order to print (not return) the value
         of `hello` as `Hello`.

         Then we pass the value of `UPPER(LEFT(@in_string, 1)) +
         LOWER(SUBSTRING(@in_string, 2, LEN(@in_string)-1))` to parameter
         `@out_string`.
 *************************************************************************** */

CREATE SCHEMA AP5;                              -- 1. creating schema `AP5` if
                                                --    not created already

```

```
208 CREATE PROCEDURE AP5.properUDP                  -- 2. creating stored procedure
209                                                 --    `AP5.properUDP`
210   @in_string VARCHAR(50)                        -- 3. declaring input parameter
211                                                 --    '@in_string` with data
212                                                 --    type `VARCHAR(50)`
213 AS
214 BEGIN                                           -- 4. beginning of executable
215                                                 --    code
216   DECLARE @out_string VARCHAR(50)               -- 5. declaring output
217                                                 --    parameter `@out_string`
218                                                 --    with same data type as
219                                                 --    input parameter
220                                                 --    `@in_string` with data
221                                                 --    type `VARCHAR(50)`
222   SET @out_string = CONCAT (                    -- 6. setting value of output
223       UPPER(LEFT(@in_string, 1)),               --    parameter `@out_string`
224       LOWER(SUBSTRING(@in_string, 2, LEN(@in_string) - 1))
225       )
226   PRINT @out_string;                            -- 7. printing (not returning)
227                                                 --    value of `@out_string`
228 END;                                            -- 8. end of executable code
229                                                 --    and stored procedure
230
231
232 /* ********************************************************************
233     3.2. In order to execute (`EXEC`) our user-defined procedure (`UDP`), we
234          must indicate the schema where it resides -- in this case, `AP5`.
235
236          Since the output is printed (displayed only) and not returned, we
237          cannot use the value by the procedure (`AP5.properUDP`).
238  ******************************************************************** */
239
240 EXEC AP5.properUDP @in_string = 'hELLO';
241
242
243 /* ********************************************************************
244     3.3. Procedures do not need input and/or output parameters if the
245          executable code does not need parameters in order to work.
246
247          In the example below, we have procedure `AP5.create_tempUDP` to create
248          database `TEMP` and prints a message when it has been completed
249          without the need of parameters.
250  ******************************************************************** */
251
252 CREATE PROCEDURE AP5.create_tempUDP             -- 1. creating stored procedure
253 AS                                              --    `AP1.create_tempUDP`
254 BEGIN                                           -- 2. beginning of procedure
255                                                 --    executable code
256   CREATE DATABASE TEMP                          -- 3. creating database `TEMP`
257   PRINT 'Database Complete'                     -- 4. message displayed (not
258                                                 --    returned)
259 END;                                            -- 5. end of executable code
```

```
260                                                 --    and stored procedure
261
262 EXEC AP5.create_tempUDP;                         -- 6. executing procedure;
263                                                 --    no parameters needed in
264                                                 --    this example
265
266 /* ************************************************************************
267     3.4. In the example below, we have procedure `AP1.drop_tempUDP` to create
268          database `TEMP` and prints a message when it has been completed
269          without the need of parameters.
270  ************************************************************************ */
271
272 CREATE PROCEDURE AP5.drop_tempUDP               -- 1. creating stored procedure
273 AS                                              --    `AP1.drop_tempUDP`
274 BEGIN                                           -- 2. beginning of executable
275                                                 --    code
276   DROP DATABASE TEMP                            -- 3. dropping database `TEMP`
277   PRINT 'Database Dropped'                      -- 4. message displayed (not
278                                                 --    returned)
279 END;                                           -- 5. end of executable code
280                                                 --    and stored procedure
281
282 EXEC AP5.drop_tempUDP;                          -- 6. executing procedure;
283                                                 --    no parameters needed in
284                                                 --    this example
285
286
287 /* ************************************************************************
288     3.5. As with all types of data objects, we can DROP procedures too.
289  ************************************************************************ */
290
291 DROP PROCEDURE AP5.create_tempUDP;               -- dropping procedure
292
293
294 /* ************************************************************************
295  4. In the two examples below, we have two procedures to change Celsius to
296     Fahrenheit and vice versa.
297
298     4.1. We first create schema `temps` in the `labs` database.
299  ************************************************************************ */
300
301 CREATE SCHEMA temps;
302
303
304 /* ************************************************************************
305     4.2. We create procedure `temps.c2f` taking in one parameter declared as a
306          FLOAT to convert temperatures in Celsius to Fahrenheit.
307  ************************************************************************ */
308
309 CREATE PROCEDURE temps.c2f @in_temp FLOAT       -- 1. input parameter
310                                                 --    initialized as a FLOAT
311 AS
```

```
312 BEGIN
313   -- formula needed (9/5 C) + 32
314   DECLARE @out_temp FLOAT                       -- 2. declaring output
315                                                 --    parameter `@out_temp`
316                                                 --    with the same datatype as
317                                                 --    `@in_temp`, in this case
318                                                 --    a FLOAT
319   SET @out_temp = (9 / 5 * @in_temp) + 32       -- 3. formula to convert
320                                                 --    Celsius to Fahrenheit
321                                                 --    including `@in_temp`
322   DECLARE @out_result VARCHAR(150)              -- 4. new output to take the
323                                                 --    the value of
324   SET @out_result = CONCAT (                    -- 5. passing values including
325       CONVERT(VARCHAR(25), @in_temp),           --    `@in_temp` (temperature
326       'C = ',                                  --    in Celsius), `@out_temp`
327       CONVERT(VARCHAR(25), @out_temp),          --    `@out_temp` (temperature
328       'F'                                      --    in Fahrenheit)
329       )
330   PRINT @out_result                             -- 6. printing value to screen
331 END;
332
333 EXEC temps.c2f 75;                              -- 7. executing procedure
334                                                 --    `temps.c2f` passing 75
335                                                 --    as temperature in Celsius
336                                                 --    returning `75C = 107F`
337
338
339 /* ***************************************************************************
340     4.3. We create procedure `temps.f2c` taking in one parameter declared as a
341          FLOAT to convert temperatures in Fahrenheit to Celsius.
342  *************************************************************************** */
343
344 CREATE PROCEDURE temps.f2c @in_temp FLOAT       -- 1. input parameter
345                                                 --    initialized as a FLOAT
346 AS
347 BEGIN
348   -- formula needed 5/9(F - 32)
349   DECLARE @out_temp FLOAT                       -- 2. declaring output
350                                                 --    parameter `@out_temp`
351                                                 --    with the same datatype as
352                                                 --    `@in_temp`, in this case
353                                                 --    a FLOAT
354   SET @out_temp = (@in_temp - 32) * 5 / 9       -- 3. formula to convert
355                                                 --    Fahrenheit to Celsius
356                                                 --    including `@in_temp`
357   DECLARE @out_result VARCHAR(150)              -- 4. new output to take the
358                                                 --    the value of
359   SET @out_result = CONCAT (                    -- 5. passing values including
360       CONVERT(VARCHAR(25), @in_temp),           --    `@in_temp` (temperature
361       'C = ',                                  --    in Fahrenheit), `@out_temp`
362       CONVERT(VARCHAR(25), @out_temp),          --    `@out_temp` (temperature
363       'F'                                      --    in Celsius)
```

```
364         )
365     PRINT @out_result                           -- 6. printing value to screen
366 END;
367
368 EXEC temps.f2c 73;                              -- 7. executing procedure
369                                                 --    `temps.f2c` passing 73
370                                                 --    as temperature in
371                                                 --    Fahrenheit returning
372                                                 --    `73F = 22.7778C`
373
374
375 /* ************************************************************************
376  5. In SQL Server, a function is a stored program that you can pass parameters
377     into and return a value.
378     https://techonthenet.com/sql_server/functions.php
379
380                     CREATE FUNCTION function_name (@input_param data_type)
381                     RETURNS data_type
382                     AS
383                     BEGIN
384                       DECLARE @output_param data_type
385                       SET @output_param = some_value
386                       executable_code
387                       RETURN output_param
388                     END;
389
390     This also means that we can take the code that we used to capitalize the
391     first letter in a string and make it into a function that we can call
392     instead of writing the same code several times and avoid the possibility of
393     errors.
394
395                     SELECT CONCAT (
396                       UPPER(LEFT(`hello`, 1)),
397                       LOWER(SUBSTRING(`hello`, 2, LEN(`hello`) - 1))
398                       );
399
400     5.1. In the example below, we create function `AP5.properUDF`.  We end the
401          name of the function with `UDF` to identify it as an user-defined
402          function.  As explained before, no two objects of the same hierarchy
403          can have the same name.  Therefore our user-defined procedure and
404          function cannot share the name (`AP5.proper`) and a suffix tells
405          the system which object to use.
406
407          The function has input parameter `@in_string` declared as VARCHAR(50)
408          -- in this case, the string `hello`.
409
410          We enclose the executable section between `BEGIN` and `END`.
411
412          We create output using parameter `@out_string`, which must have the
413          same file type as the input parameter, in order to return the value of
414          `hello` as `Hello`.
415
```

```
416          Then we pass the value of concatenation
417
418                     CONCAT(
419                       UPPER(
420                         LEFT(@in_string, 1)
421                         ),
422                       LOWER(
423                         SUBSTRING(@in_string, 2,
424                           LEN(@in_string) - 1)
425                           )
426
427          or concatenation using the `+` sign
428
429                     UPPER(
430                       LEFT(@in_string, 1)
431                       ) +
432                      LOWER(
433                        SUBSTRING(@in_string, 2,
434                         LEN(@in_string) - 1)
435                         )
436
437          to parameter `@out_string`.
438
439          As the last step, we must indicate what value must be returned from
440          the function -- in this case, parameter `@out_string`.
441   ************************************************************************ */
442
443  CREATE FUNCTION AP5.properUDF                    -- 1. creating function
444                                                   --    `AP5.properUDF`
445    (@in_string VARCHAR(50))                       -- 2. declaring input parameter
446                                                   --    `@in_string` as
447                                                   --    VARCHAR(50)
448  RETURNS VARCHAR(50)                              -- 3. indicating the same data
449                                                   --    type and size of output
450                                                   --    parameter `@out_string`
451  AS
452  BEGIN                                            -- 4. beginning of executable
453                                                   --    code
454    DECLARE @out_string VARCHAR(50)                -- 5. declaring output
455                                                   --    parameter `@out_string`
456                                                   --    with same data type as
457                                                   --    input parameter
458                                                   --    `@in_string`; same data
459                                                   --    type and size as
460                                                   --    indicated after `RETURNS`
461    SET @out_string = CONCAT (                     -- 6. setting value to output
462        UPPER(LEFT(@in_string, 1)),                --    parameter `@out_string`
463        -- LOWER(SUBSTRING(@in_string, 2, LEN(@in_string) - 1))
464        LOWER(RIGHT(@in_string, LEN(@in_string) - 1))
465        )
466    RETURN @out_string;                            -- 7. returning value of output
467                                                   --    parameter `@out_string`
```

```
468 END;                                            -- 8. end of executable code
469                                                 --    and function
470
471
472 /* ************************************************************************
473     5.2. In order to call our user-defined function (`UDF`), we must indicate
474          the schema where it resides -- in this case, `AP5`.
475  ************************************************************************ */
476
477 SELECT AP5.properUDF('hello');
478
479
480 /* ************************************************************************
481     5.3. We can use `AP5.properUDF` on any string value in any table, schema or
482          database as long as we have access to the data objects -- for example,
483          columns `AP2.Customers.FirstName` and `AP2.Customers.LastName`.  Of
484          course, first we insert values into `AP2.Customers`.
485  ************************************************************************ */
486
487 INSERT INTO AP2.Customers                       -- all values in order
488 VALUES (
489   1,
490   'Smith',
491   'John',
492   '',
493   '',
494   '',
495   '',
496   ''
497   ),
498   (
499   2,
500   'Doe',
501   'Jane',
502   '123 Main St. Apt. 1',
503   'New York',
504   'NY',
505   '10001',
506   'jane.doe@example.web'
507   );
508
509 INSERT INTO AP2.Customers (                       -- some values specifying the
510   CustomerID,                                     -- the order of the fields
511   LastName,
512   FirstName
513   )
514 VALUES (
515   3,
516   'Smith',
517   'Tom'
518   );
519
```

```
520 INSERT INTO AP2.Customers
521 VALUES (
522   5,
523   'Doe',
524   'John',
525   '',
526   'New York',
527   'NY',
528   '10001',
529   'john.doe@example.web'
530   ),
531   (
532   4,
533   'Doe',
534   'Jane',
535   '',
536   'New York',
537   'NY',
538   '',
539   'jane.doe2@example.web'
540   );
541
542 UPDATE AP2.Customers
543 SET FirstName = AP5.properUDF(FirstName),
544   LastName = AP5.properUDF(LastName);
545
546
547 /* ***************************************************************************
548     5.4. We can also create functions to FORMAT dollar amounts
549          (`AP5.dollarUDF`) and dates (`AP5.dateUDF`) considering that numeric
550          values become strings when formatted.
551  ************************************************************************** */
552
553 CREATE FUNCTION AP5.dollarUDF (@in_dollar FLOAT)
554 RETURNS VARCHAR(50)
555 AS
556 BEGIN
557   DECLARE @out_dollar VARCHAR(50)
558   SET @out_dollar = FORMAT(@in_dollar, 'c', 'en-us')
559   RETURN @out_dollar
560 END;
561
562 SELECT AP5.dollarUDF(10000) AS FormattedDollarAmout;
563
564 CREATE FUNCTION AP5.dateUDF (@in_date DATE)
565 RETURNS VARCHAR(10)
566 AS
567 BEGIN
568   DECLARE @out_date VARCHAR(10)
569   SET @out_date = FORMAT(@in_date, 'd', 'en-us')
570   RETURN @out_date
571 END;
```

```
572
573 SELECT AP5.dateUDF(GETDATE()) AS FormattedDate;
574
575
576 /* ************************************************************************
577     5.5. Going back to procedures, we can call user-defined functions inside
578          user-defined procedures (commonly referred to as stored procedures).
579          We can call `AP5.properUDF` on `AP2.Customers.FirstName` and
580          `AP2.Customers.LastName`.
581  ************************************************************************ */
582
583 CREATE PROCEDURE AP5.properUDP                  -- 1. creating stored procedure
584                                                 --    without input parameters
585 AS
586 BEGIN                                           -- 2. beginning of executable
587                                                 --    code without parameters
588   UPDATE AP2.Customers                          -- 3. updating `AP2.Customers`
589   SET FirstName = AP5.properUDF(FirstName);
590   PRINT 'Proper case assigned to first names';  -- 4. printing message
591   UPDATE AP2.Customers                          -- 5. updating `AP2.Customers`
592   SET LastName = AP5.properUDF(LastName);
593   PRINT 'Proper case assigned to last names';   -- 6. printing message
594 END;                                            -- 7. end of executable code
595                                                 --    and stored procedure
596
597 CREATE PROCEDURE AP5.CloneInvoicesUDP           -- 1. creating stored procedure
598                                                 --    `AP1.CloneInvoicesUDP`
599 AS
600 BEGIN                                           -- 2. beginning of executable
601                                                 --    code
602   DROP TABLE AP1.CloneInvoices;                 -- 3. dropping old clone table
603   PRINT 'Old table `AP1.Invoices` destroyed';   -- 4. displaying completion
604                                                 --    message
605   SELECT                                        -- 5. selecting all values from
606     InvoiceID,                                  --    `AP1.Invoices`
607     VendorID,
608     InvoiceNumber,
609     AP5.dateUDF(InvoiceDate)                    -- 6. calling date values of
610       AS InvoiceDate,                           --    columns using
611     AP5.dollarUDF(InvoiceTotal)                 --    user-defined functions
612       AS InvoiceTotal,                          --    `AP5.dateUDF` and
613     AP5.dollarUDF(PaymentTotal)                 --    `AP5.dollarUDF`
614       AS PaymentTotal,                          --
615     AP5.dollarUDF(CreditTotal)
616       AS CreditTotal,
617     TermsID,
618     AP5.dateUDF(InvoiceDueDate)
619       AS InvoiceDueDate,
620     AP5.dateUDF(PaymentDate)
621       AS PaymentDate                            -- 7. pushing values from old
622   INTO AP1.CloneInvoices                        --    table `AP1.Invoices` to
623                                                 --    new table
```

```
624                                                 --    `AP1.CloneInvoices`
625    FROM AP1.Invoices;                           -- 8. from `AP1.Invoices`
626    PRINT 'New table `AP5.Invoices` created';    -- 9. displaying completion
627                                                 --    message
628  END;                                           -- 10. end of executable code
629                                                 --     and stored procedure
630
631  EXEC AP5.CloneInvoicesUDP;
632
633
634  /* *********************************************************************
635     6. LAB #9
636        6.1. In schema `lab8` in database `labs`, create table `students`
637             (referenced as `labs.lab8.students`) with the following structure.
638
639                      student_id INT NULL
640                      student_fname VARCHAR(50) NULL
641                      student_lname VARCHAR(50) NULL
642                      student_phone VARCHAR(15) NULL
643                      student_dob DATE NULL
644                      record_date DATE NULL
645
646   ********************************************************************** */
647
648  CREATE SCHEMA lab8;
649
650  CREATE TABLE lab8.students (                   -- 1. rule of thumb: table
651                                                 --    names in plural
652    student_id INT NULL,                         -- 2. declared as INT;  can
653                                                 --    accept NULL (can have no
654                                                 --    value)
655    student_fname VARCHAR(50) NULL,              -- 3. declared as VARCHAR(50);
656                                                 --    can accept NULL (can have
657                                                 --    no value)
658    student_lname VARCHAR(50) NULL,              -- 4. declared as VARCHAR(50);
659                                                 --    can accept NULL (can have
660                                                 --    no value)
661    student_phone VARCHAR(15) NULL,              -- 5. declared as VARCHAR(50);
662                                                 --    can accept NULL (can have
663                                                 --    no value)
664    student_dob DATE NULL,                       -- 6. declared as DATE
665                                                 --
666                                                 --    DATETIME  9/20/2021 21:54
667                                                 --    DATE      9/20/2021
668                                                 --    TIME                21:54
669                                                 --
670                                                 --    can accept NULL (can have
671                                                 --    no value)
672    record_date DATE NULL                        -- 5. declared as DATE;  when
673                                                 --    record was created;  can
674                                                 --    accept NULL (can have no
675                                                 --    value)
```

```
676   );
677
678
679 /* ********************************************************************
680       6.2. Then populate the table with some data of your choice.
681
682            If we do not have a value for a specific field, we can push an empty
683            string or NULL.
684  ******************************************************************** */
685
686 INSERT INTO lab8.students
687 VALUES (
688   1,
689   'Joe',
690   'Smith',
691   '555-123-4567',
692   '1980/05/01',
693   GETDATE()                                    -- 1. built-in function to
694                                                --    retrieve system DATETIME
695   ),
696   (
697   2,
698   'Mary',
699   'Jones',
700   '212-555-1000',
701   '1983/05/16',
702   GETDATE()
703   ),
704   (
705   3,
706   'Peter',
707   'Johnson',
708   NULL,                                        -- 2. inserting empty strings
709                                                --    (``) or NULL since we
710                                                --    have no values for fields
711                                                --    to insert same number of
712                                                --    values as columns
713   '06/01/1980',
714   GETDATE()
715   );
716
717
718 /* ********************************************************************
719     6.3. In the example below, we insert only three (3) values.
720
721          We call the the three (3) corresponding columns to indicate which
722          value goes where.
723
724          We do not need to call columns in order as long order as long as
725          values are pushed in the same order (value 1 in field 1, value 2 in
726          field 2, value 3 in field 3 and value 7 in field 7).
727  ******************************************************************** */
```

```
728
729 INSERT INTO lab8.students (
730   student_id,                                   -- 1. inserting values to only
731   student_fname,                                --    four (4) columns;
732   student_lname,                                --    indicating which four (4)
733   record_date                                   --    columns
734   )
735 VALUES (
736   4,                                            -- 2. values to be inserted in
737   'Smith',                                      --    columns `student_id`,
738   'Tom',                                        --    `student_fname`,
739   GETDATE()                                     --    `student_lname` and
740   );                                            --    `record_date` receiving
741                                                 --    value from `GETDATE()`
742
743
744 /* ************************************************************************
745     6.4. In the example below, we insert row 6 before 5.
746
747          The values in `student_id` (the row identifier) are unique, but they
748          do not need to be in order.
749
750          If you need to insert values in `student_id` automatically in
751          incremental order, you would need to use `IDENTITY(1,1)` as part of
752          the table structure.  The first integer indicates that the first value
753          as 1.  The second integer indicates that the value is incremented by
754          1.  Refer to https://www.w3schools.com/sql/sql_autoincrement.asp for
755          more information.
756
757                    CREATE TABLE lab8.students (
758                      student_id INT NOT NULL IDENTITY(1, 1) PRIMARY KEY,
759                      student_fname VARCHAR(50) NULL,
760                      student_lname VARCHAR(50) NULL,
761                      student_phone VARCHAR(15) NULL,
762                      student_dob DATE NULL,
763                      record_date DATE NULL
764                      );
765  ************************************************************************ */
766
767 INSERT INTO lab8.students
768 VALUES (
769   6,
770   'John',
771   'Scott',
772   '',                                           -- 1. inserting empty strings
773   '',                                           --    (``) or NULL since we
774                                                 --    have no values for fields
775                                                 --    to insert same number of
776                                                 --    values as columns
777   GETDATE()                                     -- 2. built-in function to
778                                                 --    retrieve system DATETIME
779   ),
```

```
780    (
781    5,
782    'Mary Ann',
783    'Saunders',
784    '',                                          -- 3. inserting empty strings
785    '',                                          --    (``) or NULL since we
786                                                --    have no values for fields
787                                                --    to insert same number of
788                                                --    values as columns
789    GETDATE()                                    -- 4. built-in function to
790                                                --    retrieve system DATETIME
791    );
792
793
794  /* ************************************************************************
795      6.5. We can also delete/destroy data objects.
796
797           For the time being, we will work with tables
798           (https://techonthenet.com/sql_server/tables/drop_table.php).
799
800           Once an object is deleted, there is no way to rescue the data
801           (`ROLLBACK`) unless first creating a `SAVEPOINT`
802           (https://technet.microsoft.com/en-us/library/ms178157.aspx).
803
804           In the example below, we destroy (`DROP`) table `lab8.students`
805           understanding that, once we do, we cannot recover the structure or the
806           data.
807   ************************************************************************ */
808
809  DROP TABLE lab8.students;
810
811
812  /* ************************************************************************
813      6.6. In the case of tables, we can destroy (`TRUNCATE`) the data in the
814           table without affecting the structure of the table understanding that,
815           once we do, we cannot recover the data.
816   ************************************************************************ */
817
818  TRUNCATE TABLE lab8.students;
819
820
821  /* ************************************************************************
822      6.7. We can also modify (`ALTER`) data objects
823           (https://techonthenet.com/sql_server/tables/alter_table.php).
824
825           6.7.1. ADD     to add a column to a table
826
827           6.7.2. DROP    to delete a column to a table
828
829           6.7.3. ALTER   to change the data type or size of a column
830   ************************************************************************ */
831
```

```
832 ALTER TABLE lab8.students                       -- 1. adding new column
833 ADD Email VARCHAR(100);                         --    `Email`;  no need to
834                                                 --    specify that you are
835                                                 --    adding a column
836
837 ALTER TABLE lab8.students                       -- 2. dropping (deleting)
838 DROP COLUMN Email;                              --    column `Email` as there
839                                                 --    is no SQL statement to
840                                                 --    rename data objects;
841                                                 --    must specify that you are
842                                                 --    dropping a column
843
844 ALTER TABLE lab8.students                       -- 3. adding new (replacement)
845 ADD student_email VARCHAR(100);                 --    column `student_email`;
846                                                 --    no need to specify that
847                                                 --    you are adding a column
848
849 ALTER TABLE lab8.students                       -- 4. altering column with new
850 ALTER COLUMN student_email VARCHAR(50) NULL;    --    data type VARCHAR(50)
851                                                 --    from VARCHAR(100) and
852                                                 --    `NOT NULL`;  must specify
853                                                 --    that you are altering a
854                                                 --    column
855
856 ALTER TABLE lab8.students                       -- 5. altering column as
857 ALTER COLUMN student_id INT NOT NULL;           --    `NOT NULL`;  must specify
858                                                 --    that you are altering a
859                                                 --    column
860
861 ALTER TABLE lab8.students                       -- 6. altering column with new
862 ALTER COLUMN record_date DATETIME NOT NULL;     --    data type DATETIME
863                                                 --    from DATE and `NOT NULL`;
864                                                 --    must specify that you are
865                                                 --    altering a column
866
867 ALTER TABLE lab8.students                       -- 7. altering column with new
868 ALTER COLUMN student_fname VARCHAR(25) NOT NULL;--    data type VARCHAR(25)
869                                                 --    from VARCHAR(50) and
870                                                 --    `NOT NULL`;  must specify
871                                                 --    that you are altering a
872                                                 --    column
873
874 ALTER TABLE lab8.students                       -- 8. altering column with new
875 ALTER COLUMN student_fname VARCHAR(25) NOT NULL;--    data type VARCHAR(25)
876                                                 --    from VARCHAR(50) and
877                                                 --    `NOT NULL`;  must specify
878                                                 --    that you are altering a
879                                                 --    column
880
881 ALTER TABLE lab8.students                       -- 9. altering column with new
882 ALTER COLUMN student_id VARCHAR(5);             --    data type VARCHAR(5) from
883                                                 --    INT;  no error during
```

```
884                                                 --    conversion;  must specify
885                                                 --    that you are altering a
886                                                 --    column
887
888 ALTER TABLE lab8.students                       -- 10. altering column back to
889 ALTER COLUMN student_id INT NOT NULL;           --     data type INT from
890                                                 --     VARCHAR(5);  no error
891                                                 --     during conversion;  must
892                                                 --     specify that you are
893                                                 --     altering a column
894
895 ALTER TABLE lab8.students                       -- 11. trying to alter column
896 ALTER COLUMN student_fname FLOAT;               --     to data type FLOAT from
897                                                 --     VARCHAR(25);  conversion
898                                                 --     failure due to format
899                                                 --     incompatibility (letters
900                                                 --     to numbers)
901
902
903 /* ********************************************************************************
904     6.8. We can use `UPDATE` to write new values into an existing row.
905
906     In the example below, we UPDATE the value of column `student_phone` passing
907     value `No Number` where there is no value (`IS NULL`) or there is an empty
908     space (` `)
909  ******************************************************************************** */
910
911 UPDATE lab8.students
912 SET student_phone = 'No Number'
913 WHERE student_phone IS NULL
914   OR student_phone = '';
915
916
917 /* ********************************************************************************
918     6.9. In the example below, we UPDATE the value of column `student_email`
919          passing the value of the concatenation of `student_fname` and
920          `student_lname` with a period (`.`) between the two columns -- for
921          example, `john.smith@example.web` for `student_fname` with value of
922          `John` and `student_lname` with value of `Smith`.
923  ******************************************************************************** */
924
925 UPDATE lab8.students
926 SET student_email = LOWER(CONCAT (
927       student_fname,
928       '.',
929       student_lname,
930       '@example.web'
931       ));
932
933
934 /* ********************************************************************************
935     6.10. In the example below, we UPDATE column `record_date` where the field
```

```
936             is NULL or has an empty space (``) with value from `GETDATE()`.
937   ************************************************************************ */
938
939   UPDATE lab8.students
940   SET record_date = GETDATE()
941   WHERE record_date IS NULL
942     OR record_date = '';
943
944
945   /* ************************************************************************
946       6.10. In the example below, we can UPDATE `student_dob` to `1980/01/23`
947             where `student_id` is `1`.
948   ************************************************************************ */
949
950   UPDATE lab8.students
951   SET student_dob = '1980/01/23'
952   WHERE student_id = 1;
953
954
955   /* ************************************************************************
956      7. LAB #9
957         7.1. In schema `lab9` in database `labs`, create table `grades`
958              (referenced as `labs.lab9.grades`) with the following structure.
959
960                       grade_id INT NOT NULL UNIQUE
961                       student_id INT NOT NULL
962                       student_grade FLOAT NOT NULL
963                       grade_comment VARCHAR(255) NULL
964   ************************************************************************ */
965
966   CREATE SCHEMA lab9;
967
968   CREATE TABLE lab9.grades (
969     grade_id INT NOT NULL UNIQUE,
970     student_id INT NOT NULL,
971     student_grade FLOAT NOT NULL,
972     grade_comment VARCHAR(255) NULL
973     );
974
975
976   /* ************************************************************************
977         7.2. Then populate the table with some data of your choice.
978   ************************************************************************ */
979
980   INSERT INTO lab9.grades
981   VALUES (
982     1,
983     1,
984     80,
985     'He missed the midterm.'
986     ),
987     (
```

```
  2,
  3,
  65,
  'He slept in class.'
  ),
  (
  3,
  2,
  98,
  ''
  );


/* ***************************************************************************
     7.3. Since we have shared (`student_id`) data between `labs.lab9.grades`
          and `labs.lab8.students`, we can retrieve all the data from
          `labs.lab8.students` (main) and any related data from
          `labs.lab9.grades` (secondary) without duplicate rows (`SELECT
          DISTINCT`).

                    CREATE VIEW view_name
                    AS
                      (
                        SELECT ...
                      )
 *************************************************************************** */

SELECT DISTINCT lab8.students.student_id,
  lab8.students.student_fname,
  lab8.students.student_lname,
  lab8.students.student_phone,
  lab8.students.student_dob,
  lab8.students.record_date,
  lab9.grades.grade_id,
  -- lab9.grades.student_id AS Expr1,
  lab9.grades.student_grade,
  lab9.grades.grade_comment
FROM lab8.students
LEFT OUTER JOIN lab9.grades
  ON lab8.students.student_id = lab9.grades.student_id
ORDER BY student_lname;


/* ***************************************************************************
     7.4. Since we can query `labs.lab8.students` (main table) and
          `labs.lab9.grades` (secondary table), we can also CREATE VIEW
          `labs.lab9.students_grades_vw` from it.

          Since a VIEW calls a `SELECT` statement and is of the same hierarchy
          as a TABLE, we can query the VIEW as if it were a TABLE.
 *************************************************************************** */

```

```
1040  CREATE VIEW lab9.students_grades_vw
1041  AS
1042  SELECT DISTINCT lab8.students.student_id,
1043    lab8.students.student_fname,
1044    lab8.students.student_lname,
1045    lab8.students.student_phone,
1046    lab8.students.student_dob,
1047    lab8.students.record_date,
1048    lab9.grades.grade_id,
1049    -- lab9.grades.student_id AS Expr1,
1050    lab9.grades.student_grade,
1051    lab9.grades.grade_comment
1052  FROM lab8.students
1053  LEFT JOIN lab9.grades
1054    ON lab8.students.student_id = lab9.grades.student_id
1055  -- ORDER BY student_lname
1056
1057
1058  /* ************************************************************************
1059       7.5. Although we can UPDATE a record when we change any existing value,
1060            there are situations where we need to keep track every transaction
1061            historically -- for example, to keep track of bank transactions.  In
1062            such scenario, you should INSERT a new record for each transaction
1063            with a separate column to record the time stamp.
1064
1065            First you would need to add a column for the time stamp.
1066
1067            Then we would push the value of `GETDATE()` into the new column.  Of
1068            course, for this to work all records should have a value in new
1069            column.
1070
1071            To retrieve the latest record for student, we would need to call the
1072            `MAX()` value of all fields in the query and group the results by an
1073            identifier -- for example, `student_id` in the example below.
1074   ************************************************************************ */
1075
1076  ALTER TABLE lab9.grades                          -- adding `grade_timestamp` to
1077  ADD grade_timestamp DATETIME;                    -- table `lab9.grades`
1078
1079  UPDATE lab9.grades                               -- inserting values into
1080  SET grade_timestamp = GETDATE();                 -- `grade_timestamp`
1081
1082  INSERT INTO lab9.grades                          -- inserting two new records at
1083  VALUES (                                         -- the same time hence writing
1084    1,                                             -- the same value of
1085    1,                                             -- `GETDATE()` to both records
1086    90,
1087    'teacher''s pet'
1088    ),
1089    (
1090    5,
1091    2,
```

```
1092    85,
1093    '',
1094    GETDATE()
1095    );
1096
1097  INSERT INTO lab9.grades                          -- inserting a new record for
1098  VALUES (                                         -- for `student_id` 1
1099    1,
1100    8,
1101    95,
1102    'grade change',
1103    GETDATE()
1104    );
1105
1106  SELECT DISTINCT MAX(lab8.students.student_id) AS student_id,
1107    MAX(lab8.students.student_fname) AS student_fname,
1108    MAX(lab8.students.student_lname) AS student_lname,
1109    MAX(lab8.students.student_phone) AS student_phone,
1110    MAX(lab8.students.student_dob) AS student_dob,
1111    MAX(lab8.students.record_date) AS record_date,
1112    MAX(lab9.grades.grade_id) AS grade_id,
1113    MAX(lab9.grades.student_grade) AS student_grade,
1114    MAX(lab9.grades.grade_comment) AS grade_comment,
1115    MAX(lab9.grades.grade_timestamp) AS grade_timestamp
1116                                                   -- calling the maximum value of
1117                                                   -- `grade_timestamp` for latest
1118                                                   -- transaction of each
1119                                                   -- `lab9.grades.student_id`
1120  FROM lab9.grades
1121  INNER JOIN lab8.students
1122    ON lab9.grades.student_id = lab8.students.student_id
1123  GROUP BY lab9.grades.student_id;
1124
1125
1126  /* ***************************************************************************
1127   https://folvera.commons.gc.cuny.edu/?p=1040
1128   ************************************************************************** */
```