```
/* ************************************************************************
        CUNY ACE UPSKILLING:  INTRODUCTION TO STRUCTURED QUERY LANGUAGE
                     SF21JOB#2, 2021/11/08 to 2021/12/13
                https://folvera.commons.gc.cuny.edu/?cat=30
************************************************************************

 SESSION #8 (2021/12/01): 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.
```

```
52          https://searchwindevelopment.techtarget.com/definition/GUI
53
54    2. Now that we are going to start programability, we use parameters to pass
55       values either to a SQL script and/or receiving parameters from external
56       programs, built-in and/or user-defined procedures and/or functions.
57
58            ``In information technology, a parameter (pronounced puh-RAA-meh-tuhr,
59            from Greek for, roughly, through measure) is an item of information
60            -- such as a name, a number, or a selected option -- that is passed
61            to a program by a user or another program.  Parameters affect the
62            operation of the program receiving them.``
63            http://whatis.techtarget.com/definition/parameter
64
65            ``Parameters can be passed to the stored procedures.  This makes the
66            procedure dynamic.
67            The following points are to be noted:
68            * One or more number of parameters can be passed in a procedure.
69            * The parameter name should proceed with an @ symbol.
70            * The parameter names will be local to the procedure in which they are
71              defined.
72            The parameters are used to pass information into a procedure from the
73            line that executes the parameter.  The parameters are given just after
74            the name of the procedure on a command line.  Commas should separate
75            the list of parameters.
76            * The values can be passed to stored procedures by:
77            * By supplying the parameter values exactly in the same order as given
78              in the CREATE PROCEDURE statement.
79            * By explicitly naming the parameters and assigning the appropriate
80              value.``
81            http://devguru.com/technologies/t-sql/7132
82
83       Every time users pass values to a query commonly using a web form, you are
84       at risk of SQL injections where the user could pass a SQL statement, which
85       the server may execute.  For this reason, every database should have a
86       read-only account that queries data returning values to the front-end
87       application limiting the possibility of SQL injections and similar exploits
88       (http://searchsecurity.techtarget.com/definition/exploit).
89
90            ``SQL injection is a type of security exploit in which the attacker
91            adds Structured Query Language (SQL) code to a Web form input box to
92            gain access to resources or make changes to data.  An SQL query is a
93            request for some action to be performed on a database.  Typically, on
94            a Web form for user authentication, when a user enters their name and
95            password into the text boxes provided for them, those values are
96            inserted into a SELECT query.  If the values entered are found as
97            expected, the user is allowed access; if they aren't found, access is
98            denied.  However, most Web forms have no mechanisms in place to block
99            input other than names and passwords.  Unless such precautions are
100           taken, an attacker can use the input boxes to send their own request
101           to the database, which could allow them to download the entire
102           database or interact with it in other illicit ways.``
103           http://searchsoftwarequality.techtarget.com/definition/SQL-injection
```

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

CREATE PROCEDURE AP5.properUDP                  -- 2. creating stored procedure
```

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

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

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

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

```
416
417                     CONCAT(
418                       UPPER(
419                         LEFT(@in_string, 1)
420                         ),
421                       LOWER(
422                         SUBSTRING(@in_string, 2,
423                           LEN(@in_string) - 1)
424                           )
425
426           or concatenation using the `+` sign
427
428                     UPPER(
429                       LEFT(@in_string, 1)
430                       ) +
431                      LOWER(
432                        SUBSTRING(@in_string, 2,
433                         LEN(@in_string) - 1)
434                         )
435
436           to parameter `@out_string`.
437
438           As the last step, we must indicate what value must be returned from
439           the function -- in this case, parameter `@out_string`.
440   ********************************************************************** */
441
442  CREATE FUNCTION AP5.properUDF                    -- 1. creating function
443                                                   --    `AP5.properUDF`
444    (@in_string VARCHAR(50))                       -- 2. declaring input parameter
445                                                   --    `@in_string` as
446                                                   --    VARCHAR(50)
447  RETURNS VARCHAR(50)                              -- 3. indicating the same data
448                                                   --    type and size of output
449                                                   --    parameter `@out_string`
450  AS
451  BEGIN                                            -- 4. beginning of executable
452                                                   --    code
453    DECLARE @out_string VARCHAR(50)                -- 5. declaring output
454                                                   --    parameter `@out_string`
455                                                   --    with same data type as
456                                                   --    input parameter
457                                                   --    `@in_string`; same data
458                                                   --    type and size as
459                                                   --    indicated after `RETURNS`
460    SET @out_string = CONCAT (                     -- 6. setting value to output
461        UPPER(LEFT(@in_string, 1)),                --    parameter `@out_string`
462        -- LOWER(SUBSTRING(@in_string, 2, LEN(@in_string) - 1))
463        LOWER(RIGHT(@in_string, LEN(@in_string) - 1))
464        )
465    RETURN @out_string;                            -- 7. returning value of output
466                                                   --    parameter `@out_string`
467  END;                                             -- 8. end of executable code
```

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

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

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

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

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

```
INSERT INTO lab8.students (
  student_id,                                   -- 1. inserting values to only
  student_fname,                                --    four (4) columns;
  student_lname,                                --    indicating which four (4)
  record_date                                   --    columns
  )
VALUES (
  4,                                            -- 2. values to be inserted in
  'Smith',                                      --    columns `student_id`,
  'Tom',                                        --    `student_fname`,
  GETDATE()                                     --    `student_lname` and
  );                                            --    `record_date` receiving
                                                --    value from `GETDATE()`


/* ************************************************************************
    6.4. In the example below, we insert row 6 before 5.

         The values in `student_id` (the row identifier) are unique, but they
         do not need to be in order.

         If you need to insert values in `student_id` automatically in
         incremental order, you would need to use `IDENTITY(1,1)` as part of
         the table structure.  The first integer indicates that the first value
         as 1.  The second integer indicates that the value is incremented by
         1.  Refer to https://www.w3schools.com/sql/sql_autoincrement.asp for
         more information.

                    CREATE TABLE lab8.students (
                      student_id INT NOT NULL IDENTITY(1, 1) PRIMARY KEY,
                      student_fname VARCHAR(50) NULL,
                      student_lname VARCHAR(50) NULL,
                      student_phone VARCHAR(15) NULL,
                      student_dob DATE NULL,
                      record_date DATE NULL
                      );
 ************************************************************************ */

INSERT INTO lab8.students
VALUES (
  6,
  'John',
  'Scott',
  '',                                           -- 1. inserting empty strings
  '',                                           --    (``) or NULL since we
                                                --    have no values for fields
                                                --    to insert same number of
                                                --    values as columns
  GETDATE()                                     -- 2. built-in function to
                                                --    retrieve system DATETIME
  ),
  (
```

```
780    5,
781    'Mary Ann',
782    'Saunders',
783    '',                                          -- 3. inserting empty strings
784    '',                                          --    (``) or NULL since we
785                                                 --    have no values for fields
786                                                 --    to insert same number of
787                                                 --    values as columns
788    GETDATE()                                    -- 4. built-in function to
789                                                 --    retrieve system DATETIME
790    );
791
792
793  /* ************************************************************************
794      6.5. We can also delete/destroy data objects.
795
796           For the time being, we will work with tables
797           (https://techonthenet.com/sql_server/tables/drop_table.php).
798
799           Once an object is deleted, there is no way to rescue the data
800           (`ROLLBACK`) unless first creating a `SAVEPOINT`
801           (https://technet.microsoft.com/en-us/library/ms178157.aspx).
802
803           In the example below, we destroy (`DROP`) table `lab8.students`
804           understanding that, once we do, we cannot recover the structure or the
805           data.
806   ************************************************************************ */
807
808  DROP TABLE lab8.students;
809
810
811  /* ************************************************************************
812      6.6. In the case of tables, we can destroy (`TRUNCATE`) the data in the
813           table without affecting the structure of the table understanding that,
814           once we do, we cannot recover the data.
815   ************************************************************************ */
816
817  TRUNCATE TABLE lab8.students;
818
819
820  /* ************************************************************************
821      6.7. We can also modify (`ALTER`) data objects
822           (https://techonthenet.com/sql_server/tables/alter_table.php).
823
824           6.7.1. ADD     to add a column to a table
825
826           6.7.2. DROP    to delete a column to a table
827
828           6.7.3. ALTER   to change the data type or size of a column
829   ************************************************************************ */
830
831  ALTER TABLE lab8.students                      -- 1. adding new column
```

```
ADD Email VARCHAR(100);                          --    `Email`;  no need to
                                                 --    specify that you are
                                                 --    adding a column

ALTER TABLE lab8.students                        -- 2. dropping (deleting)
DROP COLUMN Email;                               --    column `Email` as there
                                                 --    is no SQL statement to
                                                 --    rename data objects;
                                                 --    must specify that you are
                                                 --    dropping a column

ALTER TABLE lab8.students                        -- 3. adding new (replacement)
ADD student_email VARCHAR(100);                  --    column `student_email`;
                                                 --    no need to specify that
                                                 --    you are adding a column

ALTER TABLE lab8.students                        -- 4. altering column with new
ALTER COLUMN student_email VARCHAR(50) NULL;     --    data type VARCHAR(50)
                                                 --    from VARCHAR(100) and
                                                 --    `NOT NULL`;  must specify
                                                 --    that you are altering a
                                                 --    column

ALTER TABLE lab8.students                        -- 5. altering column as
ALTER COLUMN student_id INT NOT NULL;            --    `NOT NULL`;  must specify
                                                 --    that you are altering a
                                                 --    column

ALTER TABLE lab8.students                        -- 6. altering column with new
ALTER COLUMN record_date DATETIME NOT NULL;      --    data type DATETIME
                                                 --    from DATE and `NOT NULL`;
                                                 --    must specify that you are
                                                 --    altering a column

ALTER TABLE lab8.students                        -- 7. altering column with new
ALTER COLUMN student_fname VARCHAR(25) NOT NULL;--    data type VARCHAR(25)
                                                 --    from VARCHAR(50) and
                                                 --    `NOT NULL`;  must specify
                                                 --    that you are altering a
                                                 --    column

ALTER TABLE lab8.students                        -- 8. altering column with new
ALTER COLUMN student_fname VARCHAR(25) NOT NULL;--    data type VARCHAR(25)
                                                 --    from VARCHAR(50) and
                                                 --    `NOT NULL`;  must specify
                                                 --    that you are altering a
                                                 --    column

ALTER TABLE lab8.students                        -- 9. altering column with new
ALTER COLUMN student_id VARCHAR(5);              --    data type VARCHAR(5) from
                                                 --    INT;  no error during
                                                 --    conversion;  must specify
```

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

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

CREATE VIEW lab9.students_grades_vw
```

```
1040  AS
1041  SELECT DISTINCT lab8.students.student_id,
1042    lab8.students.student_fname,
1043    lab8.students.student_lname,
1044    lab8.students.student_phone,
1045    lab8.students.student_dob,
1046    lab8.students.record_date,
1047    lab9.grades.grade_id,
1048    -- lab9.grades.student_id AS Expr1,
1049    lab9.grades.student_grade,
1050    lab9.grades.grade_comment
1051  FROM lab8.students
1052  LEFT JOIN lab9.grades
1053    ON lab8.students.student_id = lab9.grades.student_id
1054  -- ORDER BY student_lname
1055
1056
1057  /* ************************************************************************
1058       7.5. Although we can UPDATE a record when we change any existing value,
1059            there are situations where we need to keep track every transaction
1060            historically -- for example, to keep track of bank transactions.  In
1061            such scenario, you should INSERT a new record for each transaction
1062            with a separate column to record the time stamp.
1063
1064            First you would need to add a column for the time stamp.
1065
1066            Then we would push the value of `GETDATE()` into the new column.  Of
1067            course, for this to work all records should have a value in new
1068            column.
1069
1070            To retrieve the latest record for student, we would need to call the
1071            `MAX()` value of all fields in the query and group the results by an
1072            identifier -- for example, `student_id` in the example below.
1073   ************************************************************************ */
1074
1075  ALTER TABLE lab9.grades                          -- adding `grade_timestamp` to
1076  ADD grade_timestamp DATETIME;                    -- table `lab9.grades`
1077
1078  UPDATE lab9.grades                               -- inserting values into
1079  SET grade_timestamp = GETDATE();                 -- `grade_timestamp`
1080
1081  INSERT INTO lab9.grades                          -- inserting two new records at
1082  VALUES (                                         -- the same time hence writing
1083    1,                                             -- the same value of
1084    1,                                             -- `GETDATE()` to both records
1085    90,
1086    'teacher''s pet'
1087    ),
1088    (
1089    5,
1090    2,
1091    85,
```

```
1092    '',
1093    GETDATE()
1094    );
1095
1096  INSERT INTO lab9.grades                          -- inserting a new record for
1097  VALUES (                                         -- for `student_id` 1
1098    1,
1099    8,
1100    95,
1101    'grade change',
1102    GETDATE()
1103    );
1104
1105  SELECT DISTINCT MAX(lab8.students.student_id) AS student_id,
1106    MAX(lab8.students.student_fname) AS student_fname,
1107    MAX(lab8.students.student_lname) AS student_lname,
1108    MAX(lab8.students.student_phone) AS student_phone,
1109    MAX(lab8.students.student_dob) AS student_dob,
1110    MAX(lab8.students.record_date) AS record_date,
1111    MAX(lab9.grades.grade_id) AS grade_id,
1112    MAX(lab9.grades.student_grade) AS student_grade,
1113    MAX(lab9.grades.grade_comment) AS grade_comment,
1114    MAX(lab9.grades.grade_timestamp) AS grade_timestamp
1115                                                   -- calling the maximum value of
1116                                                   -- `grade_timestamp` for latest
1117                                                   -- transaction of each
1118                                                   -- `lab9.grades.student_id`
1119  FROM lab9.grades
1120  INNER JOIN lab8.students
1121    ON lab9.grades.student_id = lab8.students.student_id
1122  GROUP BY lab9.grades.student_id;
1123
1124
1125  /* ***************************************************************************
1126   https://folvera.commons.gc.cuny.edu/?p=1044
1127   ************************************************************************** */
```