```
/* ************************************************************************
          INTRODUCTION TO STRUCTURED QUERY LANGUAGE FOR DATA ANALYTICS
                    WS23SQL1001, 2023/04/03 to 2023/05/03
                https://folvera.commons.gc.cuny.edu/?cat=33
 ************************************************************************

  SESSION #8 (2023/04/26): CREATING DATABASE OBJECTS

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

 1. LAB #7
    Write a query without duplicate rows (`SELECT DISTINCT`)
    1.01. to get all shared values from tables `AP1.InvoiceLineItems` and
          `AP1.GLAccounts` (`INNER JOIN`),
    1.02. adding today's date as `TodaysDate` formatted as short date
    1.03. where `AP1.GLAccounts.AccountDescription` starts with `book`
          (`AP1.GLAccounts.AccountDescription LIKE('book%')`) and
          `AP1.InvoiceLineItems.InvoiceLineItemAmount` is at least 1000.00
          (inclusive) -- first condition composed of two conditions
    1.04. or where `AP1.GLAccounts.AccountDescription` contains `mail` and
          `AP1.InvoiceLineItems.InvoiceLineItemAmount` is no more than
          100.00 (inclusive) -- second condition composed of two conditions
          (second condition in parenthesis (OR secondary_codition1 AND
          secondary_condition2))
    1.05. ordered first by `AP1.GLAccounts.AccountDescription` and then by
          `AP1.InvoiceLineItems.InvoiceLineItemAmount`.
    1.06. Then make a view in schema `lab07` in database `labs`.
 ************************************************************************ */

SELECT DISTINCT AP1.InvoiceLineItems.InvoiceID,
  AP1.InvoiceLineItems.InvoiceSequence,
  AP1.InvoiceLineItems.AccountNo,
  AP1.InvoiceLineItems.InvoiceLineItemAmount,
  AP1.InvoiceLineItems.InvoiceLineItemDescription,
  -- AP1.GLAccounts.AccountNo AS Expr1,
  AP1.GLAccounts.AccountDescription
  /*,
  FORMAT(GETDATE(), 'd', 'en-us') AS TodaysDate*/
FROM AP1.InvoiceLineItems
INNER JOIN AP1.GLAccounts
  ON AP1.InvoiceLineItems.AccountNo = AP1.GLAccounts.AccountNo
WHERE
  (                                          -- 1. first block of two
                                             --    conditions that must be
                                             --    true
    AP1.GLAccounts.AccountDescription LIKE 'book%'
    AND AP1.InvoiceLineItems.InvoiceLineItemAmount >= 1000
    )
  OR                                         -- 2. `OR` to indicate that
```

```
                                                --    either the first block
                                                --    (above) or the second
                                                --    (below) must be true
   (                                            -- 3. second block of two
                                                --    conditions that must be
                                                --    true
     AP1.GLAccounts.AccountDescription LIKE '%mail%'
     AND AP1.InvoiceLineItems.InvoiceLineItemAmount <= 100
     )
ORDER BY AP1.GLAccounts.AccountDescription,
  AP1.InvoiceLineItems.InvoiceLineItemAmount,
  AP1.InvoiceLineItems.InvoiceID,
  AP1.InvoiceLineItems.InvoiceSequence,
  AP1.InvoiceLineItems.AccountNo,
  AP1.InvoiceLineItems.InvoiceLineItemDescription;


/* ************************************************************************
          At this point, we make a view in schema `lab07` in database `labs`
          excluding the `ORDER BY` clause as it cannot be used when creating
          views.
 ************************************************************************ */

CREATE VIEW lab07.InvoiceLineItemsGLAccountsVW
AS
SELECT DISTINCT WS23SQL1001.AP1.InvoiceLineItems.InvoiceID,
  WS23SQL1001.AP1.InvoiceLineItems.InvoiceSequence,
  WS23SQL1001.AP1.InvoiceLineItems.AccountNo,
  WS23SQL1001.AP1.InvoiceLineItems.InvoiceLineItemAmount,
  WS23SQL1001.AP1.InvoiceLineItems.InvoiceLineItemDescription,
  -- WS23SQL1001.AP1.GLAccounts.AccountNo AS Expr1,
  WS23SQL1001.AP1.GLAccounts.AccountDescription
FROM WS23SQL1001.AP1.InvoiceLineItems
INNER JOIN WS23SQL1001.AP1.GLAccounts
  ON WS23SQL1001.AP1.InvoiceLineItems.AccountNo =
    WS23SQL1001.AP1.GLAccounts.AccountNo
WHERE (
    WS23SQL1001.AP1.GLAccounts.AccountDescription LIKE 'book%'
    AND WS23SQL1001.AP1.InvoiceLineItems.InvoiceLineItemAmount >= 1000
    )
  OR (
    WS23SQL1001.AP1.GLAccounts.AccountDescription LIKE '%mail%'
    AND WS23SQL1001.AP1.InvoiceLineItems.InvoiceLineItemAmount <= 100
    );


/* ************************************************************************
          As an alternative, we can use an alias for each table.

                    `i` for `AP1.InvoiceLineItems`
                    `g` for `AP1.GLAccounts`
 ************************************************************************ */
```

```sql

CREATE VIEW lab07.InvoiceLineItemsGLAccountsVW
AS
SELECT DISTINCT i.InvoiceID,
  i.InvoiceSequence,
  i.AccountNo,
  i.InvoiceLineItemAmount,
  i.InvoiceLineItemDescription,
  g.AccountDescription
FROM WS23SQL1001.AP1.InvoiceLineItems AS i
INNER JOIN WS23SQL1001.AP1.GLAccounts AS g
  ON i.AccountNo = g.AccountNo
WHERE (
    g.AccountDescription LIKE 'book%'
    AND i.InvoiceLineItemAmount >= 1000
    )
  OR (
    g.AccountDescription LIKE '%mail%'
    AND i.InvoiceLineItemAmount <= 100
    );


/* ***************************************************************************
 2. LAB #8 (CREATING OBJECTS)
    2.01. Create database `labs`.
    2.02. Create schema `lab08` in database `labs`.
    2.03. Create table `my_family` in schema `lab08` with the following
          structure  choosing the best file type for each column and assign
          `NOT NULL` to each.

                    row_id
                    person_fname
                    person_lname
                    relation

    2.04. Insert values accordingly.
    2.05. Modify table `my_family` to add a column `dob`.
    2.06. Update the table with data in `dob` (new values in an existing record
          in table `labs.lab08.my_family`).
    2.07. Change column `dob` to `NOT NULL`.
 *************************************************************************** */

CREATE DATABASE labs;                           -- 1. creating database `labs`
                                                --    1.1. run #1 (all `CREATE
                                                --         DATABASE` statements
                                                --         run together, but
                                                --         separately from
                                                --         other statements)

CREATE SCHEMA lab08;                            -- 2. creating schema `lab08`

CREATE TABLE lab08.my_family (                  -- 3. creating table
```

```
157    row_id INT NOT NULL,                          --    `lab08.my_family`
158    person_fname VARCHAR(25) NOT NULL,            --    3.1. run #3 (all `CREATE
159    person_lname VARCHAR(25) NOT NULL,            --         TABLE` statement run
160    relation VARCHAR(15) NOT NULL                 --         together, but
161    );                                            --         separately from
162                                                  --         other statements)
163
164  INSERT INTO lab08.my_family                     -- 4. inserting new values into
165  VALUES (                                        --    table `lab08.my_family`
166    1,                                            --    4.1. each row/record
167    'John',                                       --         within a set of
168    'Doe',                                        --         parenthesis followed
169    'crazy uncle'                                 --         by a comma between
170    ),                                            --         rows/records
171    (                                             --    4.2. run #4 (all `INSERT`
172    2,                                            --         statements run
173    'Michael',                                    --         together, separately
174    'Jones',                                      --         from other
175    'cousin'                                      --         statements)
176    ),
177    (
178    3,
179    'Lucy',
180    'Smith',
181    'aunt'
182    );
183
184  ALTER TABLE lab08.my_family                     -- 5. altering table
185  ADD dob DATE;                                   --    `lab08.my_family` to add
186                                                  --    column `dob` with data
187                                                  --    type `DATE`
188                                                  --    5.1. run #5 (all `ALTER`
189                                                  --         statements run
190                                                  --         together, separately
191                                                  --         from other
192                                                  --         statements)
193
194  UPDATE lab08.my_family                          -- 6. updating table
195  SET dob = '1970-01-01'                          --    `lab08.my_family` to pass
196  WHERE row_id = 1;                               --    a new values to column
197                                                  --    `dob` in the existing
198  UPDATE lab08.my_family                          --    table `lab08.my_family`
199  SET dob = '1980/05/09'                          --    6.1. run #6 (all `UPDATE`
200  WHERE row_id = 2;                               --         statements run
201                                                  --         together, separately
202  UPDATE lab08.my_family                           --         from other
203  SET dob = '1988/08/19'                          --         statements)
204  WHERE row_id = 3;
205
206  ALTER TABLE lab08.my_family                     -- 7. changing new column `dob`
207  ALTER COLUMN dob DATE NOT NULL;                 --    to `NOT NULL` as column
208                                                  --    now has values
```

```
209                                                 --    7.1. run #7 (this `ALTER`
210                                                 --         statement run after
211                                                 --         populating new
212                                                 --         column `dob`
213
214
215 /* ***************************************************************************
216  3. LAB #9 (ALTERING OBJECTS)
217     Make some changes to `AP1.ContactUpdates` and `AP1.Vendors`.
218
219     3.01. Add column `Email` to `AP1.ContactUpdates`, which should be
220           `VARCHAR(100)` and `NOT NULL` (HINT: `UPDATE` first, then `NOT NULL`
221           since a new column has no values).
222
223           First we need to add the column to the table.
224  *************************************************************************** */
225
226 ALTER TABLE AP1.ContactUpdates
227 ADD Email VARCHAR(100);
228
229
230 /* ***************************************************************************
231     3.02. Populate the column (every field).
232
233           If we use `LastName` as part of the email, we should remove the
234           apostrophe in `O'Sullivan`.  Make sure to push the new values to an
235           existing row in lower case (HINT: `UPDATE`).
236
237           Note below the changes that happen when using `REPLACE()` to change
238           certain characters to empty strings in order to make the email
239           accounts using `VendorName`.
240
241           Note that, if `REPLACE()` does not find the string that we ask the
242           function to replace, the value in unchanged.  In the email shown
243           below, there is no ampersand (`&`) so the value is unchanged in the
244           third (3rd) instance of `REPLACE()` before it continues to the next
245           instance.  The same happens in the fifth (5th) instance when
246           searching for apostrophes (`''`, called with two single quotes as an
247           escape character).
248  *************************************************************************** */
249
250 UPDATE AP1.ContactUpdates
251 SET Email=LOWER(
252             REPLACE(
253               REPLACE(
254                 REPLACE(
255                   REPLACE(
256                     CONCAT (
257                       FirstName,
258                       '.',
259                       LastName,
260                       '@',
```

```
                    VendorName,
                    '.foo'
                ),      -- 1. `Anthony.Antavius@Courier Companies, Inc.foo`
              ' ', ''), -- 2. `Anthony.Antavius@CourierCompanies,Inc.foo`
            '&', ''),   -- 3. `Anthony.Antavius@CourierCompanies,Inc.foo`
          ',', ''),     -- 4. `Anthony.Antavius@CourierCompaniesInc.foo`
        '''', '')       -- 5. `anthony.antavius@couriercompaniesinc.foo`
      )                 -- 6. `anthony.antavius@couriercompaniesinc.foo`
FROM AP1.ContactUpdates
INNER JOIN AP1.Vendors
  ON AP1.ContactUpdates.VendorID = AP1.Vendors.VendorID;
FROM AP1.ContactUpdates
INNER JOIN AP1.Vendors
  ON AP1.ContactUpdates.VendorID = AP1.Vendors.VendorID;


/* ***************************************************************************
      Since record with `VendorID` 76 does not have a related `VendorName`
      (hence no means to make an email), we can assign a value.  In this
      case, We can assign `NO EMAIL` to that record.
 *************************************************************************** */

UPDATE AP1.ContactUpdates
SET Email = 'NO EMAIL'
WHERE VendorID = 76


/* ***************************************************************************
      At this point, we change the column to `NOT NULL`.
 *************************************************************************** */

ALTER TABLE AP1.ContactUpdates
ALTER COLUMN Email VARCHAR(100) NOT NULL;


/* ***************************************************************************
    3.03. Add column `VendorAddress` to `AP1.Vendors`, which should be
      `VARCHAR(150)` and `NOT NULL`.
 *************************************************************************** */

ALTER TABLE AP1.Vendors
ADD VendorAddress VARCHAR(150);


/* ***************************************************************************
      Then we move the values of `VendorAddress1` and `VendorAddress2` to
      `VendorAddress`.
 *************************************************************************** */

UPDATE AP1.Vendors
SET VendorAddress = CONCAT (
    VendorAddress1,
```

```
313      ' ',
314      VendorAddress2
315      );
316
317
318  /* ***************************************************************************
319            Then we make sure the new column has the data and delete the original
320            two columns.
321   *************************************************************************** */
322
323  ALTER TABLE AP1.Vendors
324  DROP COLUMN VendorAddress1;
325
326  ALTER TABLE AP1.Vendors
327  DROP COLUMN VendorAddress2;
328
329
330  /* ***************************************************************************
331            Then we change the new column to `NOT NULL`.
332   *************************************************************************** */
333
334  ALTER TABLE AP1.Vendors
335  ALTER COLUMN VendorAddress VARCHAR(150) NOT NULL;
336
337
338  /* ***************************************************************************
339      3.04. Call all the values from `AP1.ContactUpdates` with any corresponding
340            values in `AP1.Vendors` (HINT: `LEFT JOIN` to get 8 records).
341   *************************************************************************** */
342
343  SELECT DISTINCT *
344  FROM AP1.ContactUpdates
345  LEFT JOIN AP1.Vendors
346    ON AP1.ContactUpdates.VendorID = AP1.Vendors.VendorID;
347
348
349  /* ***************************************************************************
350      3.05. Make a view named `AP1.ContactUpdatesVendorsVW` from the prior query.
351   *************************************************************************** */
352
353  CREATE VIEW AP1.ContactUpdatesVendorsVW
354  AS
355  (
356      SELECT DISTINCT AP1.ContactUpdates.VendorID,
357        AP1.ContactUpdates.LastName,
358        AP1.ContactUpdates.FirstName,
359        AP1.ContactUpdates.Email,
360        -- AP1.Vendors.VendorID AS Expr1,
361        AP1.Vendors.VendorName,
362        AP1.Vendors.VendorCity,
363        AP1.Vendors.VendorState,
364        AP1.Vendors.VendorZipCode,
```

```
365       AP1.Vendors.VendorPhone,
366       AP1.Vendors.VendorContactLName,
367       AP1.Vendors.VendorContactFName,
368       AP1.Vendors.DefaultTermsID,
369       AP1.Vendors.DefaultAccountNo,
370       AP1.Vendors.VendorAddress
371     FROM AP1.ContactUpdates
372     LEFT JOIN AP1.Vendors
373       ON AP1.ContactUpdates.VendorID = AP1.Vendors.VendorID
374     );
375
376
377  /* ***************************************************************************
378           As an alternative, we can use an alias for each table.
379
380                     `c` for `AP1.ContactUpdates`
381                     `v` for `AP1.Vendors`
382   ************************************************************************** */
383
384  ALTER VIEW AP1.ContactUpdatesVendorsVW
385  AS
386  (
387      SELECT DISTINCT c.VendorID,
388        c.LastName,
389        c.FirstName,
390        c.Email,
391        v.VendorName,
392        v.VendorCity,
393        v.VendorState,
394        v.VendorZipCode,
395        v.VendorPhone,
396        v.VendorContactLName,
397        v.VendorContactFName,
398        v.DefaultTermsID,
399        v.DefaultAccountNo,
400        v.VendorAddress
401      FROM AP1.ContactUpdates AS c
402      LEFT JOIN AP1.Vendors AS v
403        ON c.VendorID = v.VendorID
404      );
405
406
407  /* ***************************************************************************
408           We can also make the view in schema `lab09` database `labs`.
409   ************************************************************************** */
410
411  CREATE SCHEMA lab09;
412
413  CREATE VIEW lab09.ContactUpdatesVendorsVW
414  AS
415  (
416      SELECT DISTINCT
```

```
417       WS23SQL1001.AP1.ContactUpdates.VendorID,    -- calling database
418       WS23SQL1001.AP1.ContactUpdates.LastName,    -- `WS23SQL1001` since the view
419       WS23SQL1001.AP1.ContactUpdates.FirstName,   -- is in data `labs`
420       WS23SQL1001.AP1.ContactUpdates.Email,
421       -- WS23SQL1001.AP1.Vendors.VendorID AS Expr1,
422       WS23SQL1001.AP1.Vendors.VendorName,
423       WS23SQL1001.AP1.Vendors.VendorCity,
424       WS23SQL1001.AP1.Vendors.VendorState,
425       WS23SQL1001.AP1.Vendors.VendorZipCode,
426       WS23SQL1001.AP1.Vendors.VendorPhone,
427       WS23SQL1001.AP1.Vendors.VendorContactLName,
428       WS23SQL1001.AP1.Vendors.VendorContactFName,
429       WS23SQL1001.AP1.Vendors.DefaultTermsID,
430       WS23SQL1001.AP1.Vendors.DefaultAccountNo,
431       WS23SQL1001.AP1.Vendors.VendorAddress
432     FROM WS23SQL1001.AP1.ContactUpdates
433     LEFT JOIN WS23SQL1001.AP1.Vendors
434       ON WS23SQL1001.AP1.ContactUpdates.VendorID =
435         WS23SQL1001.AP1.Vendors.VendorID
436     );
437
438 /* ************************************************************************
439           As an alternative, we can use an alias for each table.
440
441                     `c` for `WS23SQL1001.AP1.ContactUpdates`
442                     `v` for `WS23SQL1001.AP1.Vendors`
443
444           In this scenario, using an alias for each table can make the creating
445           the same view in other databases by just changing the names of the
446           tables rather than adding the database before each column.
447  ************************************************************************ */
448
449 ALTER VIEW lab09.ContactUpdatesVendorsVW
450 AS
451 (
452     SELECT DISTINCT c.VendorID,
453       c.LastName,
454       c.FirstName,
455       c.Email,
456       v.VendorName,
457       v.VendorCity,
458       v.VendorState,
459       v.VendorZipCode,
460       v.VendorPhone,
461       v.VendorContactLName,
462       v.VendorContactFName,
463       v.DefaultTermsID,
464       v.DefaultAccountNo,
465       v.VendorAddress
466     FROM WS23SQL1001.AP1.ContactUpdates AS c     -- quick change, adding the
467     LEFT JOIN WS23SQL1001.AP1.Vendors AS v       -- database name in the `FROM`
468       ON c.VendorID = v.VendorID                 -- clause rather than placing
```

```
469     );                                          -- the name of the database
470                                                 -- before each column
471
472
473 /* *********************************************************************
474  4. The following set of concepts that is good for we to know involve how
475     humans communicate with the computer and vice versa.
476
477          ``A command line interface (CLI) is a text-based user interface (UI)
478          used to view and manage computer files.  Command line interfaces are
479          also called command-line user interfaces, console user interfaces and
480          character user interfaces...
481          Before the mouse, users interacted with an operating system (OS) or
482          application with a keyboard. Users typed commands in the command line
483          interface to run tasks on a computer.
484          Typically, the command line interface features a black box with white
485          text.  The user responds to a prompt in the command line interface by
486          typing a command.  The output or response from the system can include
487          a message, table, list, or some other confirmation of a system or
488          application action.
489          Today, most users prefer the graphical user interface (GUI) offered by
490          operating systems such as Windows, Linux and macOS.  Most current
491          Unix-based systems offer both a command line interface and a graphical
492          user interface.
493          The MS-DOS operating system and the command shell in the Windows
494          operating system are examples of command line interfaces.  In
495          addition, programming languages can support command line interfaces,
496          such as Python.``
497          https://searchwindowsserver.techtarget.com/definition/command-line-     ↵
              interface-CLI
498
499          ``A GUI (usually pronounced GOO-ee) is a graphical (rather than purely
500          textual) user interface to a computer.  As we read this, we are
501          looking at the GUI or graphical user interface of your particular Web
502          browser.  The term came into existence because the first interactive
503          user interfaces to computers were not graphical; they were
504          text-and-keyboard oriented and usually consisted of commands we had
505          to remember and computer responses that were infamously brief.  The
506          command interface of the DOS operating system (which we can still get
507          to from your Windows operating system) is an example of the typical
508          user-computer interface before GUIs arrived.  An intermediate step in
509          user interfaces between the command line interface and the GUI was the
510          non-graphical menu-based interface, which let we interact by using a
511          mouse rather than by having to type in keyboard commands.
512          https://searchwindevelopment.techtarget.com/definition/GUI
513
514  5. Now that we are going to start programability, we use parameters to pass
515     values either to a SQL script and/or receiving parameters from external
516     programs, built-in and/or user-defined procedures and/or functions.
517
518          ``In information technology, a parameter (pronounced puh-RAA-meh-tuhr,
519          from Greek for, roughly, through measure) is an item of information
```

```
520           -- such as a name, a number, or a SELECT DISTINCTed option -- that is
               passed
521           to a program by a user or another program.  Parameters affect the
522           operation of the program receiving them.``
523           http://whatis.techtarget.com/definition/parameter
524
525           ``Parameters can be passed to the stored procedures.  This makes the
526           procedure dynamic.
527           The following points are to be noted:
528           * One or more number of parameters can be passed in a procedure.
529           * The parameter name should proceed with an @ symbol.
530           * The parameter names will be local to the procedure in which they are
531             defined.
532           The parameters are used to pass information into a procedure from the
533           line that executes the parameter.  The parameters are given just after
534           the name of the procedure on a command line.  Commas should separate
535           the list of parameters.
536           * The values can be passed to stored procedures by:
537           * By supplying the parameter values exactly in the same order as given
538             in the CREATE PROCEDURE statement.
539           * By explicitly naming the parameters and assigning the appropriate
540             value.``
541           http://devguru.com/technologies/t-sql/7132
542
543      Every time users pass values to a query commonly using a web form, we are
544      at risk of SQL injections where the user could pass a SQL statement, which
545      the server may execute.  For this reason, every database should have a
546      read-only account that queries data returning values to the front-end
547      application limiting the possibility of SQL injections and similar exploits
548      (http://searchsecurity.techtarget.com/definition/exploit).
549
550           ``SQL injection is a type of security exploit in which the attacker
551           adds Structured Query Language (SQL) code to a Web form input box to
552           gain access to resources or make changes to data.  An SQL query is a
553           request for some action to be performed on a database.  Typically, on
554           a Web form for user authentication, when a user enters their name and
555           password into the text boxes provided for them, those values are
556           inserted into a SELECT DISTINCT query.  If the values entered are found
               as
557           expected, the user is allowed access; if they aren't found, access is
558           denied.  However, most Web forms have no mechanisms in place to block
559           input other than names and passwords.  Unless such precautions are
560           taken, an attacker can use the input boxes to send their own request
561           to the database, which could allow them to download the entire
562           database or interact with it in other illicit ways.``
563           http://searchsoftwarequality.techtarget.com/definition/SQL-injection
564
565      5.01. In the example below, we first declare parameter `@foo` and `@pi`
566            (always starting with the `@` sign) to tell SQL Server that it should
567            expect a value within SQL script and/or receiving a value from an
568            external programs, built-in and/or user-defined procedure and/or
569            function.
```

```
570
571           Note that there are no limit to the number of parameters we can use
572           in a SQL script.
573
574           Go to https://en.wikipedia.org/wiki/Foobar if we are interested about
575           terms `foobar`, `foo` and `bar`.
576
577           We first declare parameters `@foo` with data type INT and `@pi` with
578           data type FLOAT.
579
580           We then set (assign) a value to parameters `@foo` and `@pi`.
581
582           We can then call the value of `@foo` and `@pi` within a SQL statement.
583  ************************************************************************** */
584
585  DECLARE @foo INT = 3,                          -- 1. declaring parameter
586                                                 --    `@foo` as INT initialized
587                                                 --    with value of 3
588    @pi FLOAT = 3.14159265358979;                -- 2. declaring parameter `@pi`
589                                                 --    as FLOAT initialized with
590                                                 --    value of 3.14159265358979
591
592
593  /* ************************************************************************
594      5.02. At this point, we can call the values of parameters `@foo` and `@pi`
595           in a SQL statement.
596
597                     +-------+------------------+------------------+
598                     | foo   | pi(e)            | foo pi(e)        |
599                     +-------+------------------+------------------+
600                     | 3     | 3.14159265358979 | 9.42477796076937 |
601                     +-------+------------------+------------------+
602
603           Note that we use a `SELECT DISTINCT` statement in the same way we use ↵
             `PRINT`
604           in other languages to show the output in the console.
605  ************************************************************************** */
606
607  SELECT DISTINCT @foo AS 'foo',                       -- 1. displaying value ↵
     of
608                                                 --    `@foo` (INT) with alias
609                                                 --    `foo`
610    @pi AS 'pi(e)',                              -- 2. displaying value of
611                                                 --    `@pi` (FLOAT) with alias
612                                                 --    `pi(e)`
613    @foo * @pi AS 'foo pi(e)';                   -- 3. displaying value of
614                                                 --    `@foo` multiplied by
615                                                 --    `@pi`, which returns a
616                                                 --    FLOAT due to data type
617                                                 --    conversion with alias
618                                                 --    `foo pi(e)`
619
```

```
620
621 /* ************************************************************************
622  6. ``In SQL Server, a procedure is a stored program that you can pass
623     parameters into.  It does not return a value like a function does.
624     However, it can return a success/failure status to the procedure that
625     called it.``
626     https://techonthenet.com/sql_server/procedures.php
627
628                     CREATE PROCEDURE procedure_name [@input_param data_type]
629                     AS
630                     BEGIN
631                       [DECLARE @output_param data_type
632                       SET @output_param = some_value]
633                       executable_code
634                     END;
635
636     This means that we can take the code that we used to capitalize the first
637     letter in a string and make it into a procedure that we can call indicating
638     the input parameter instead of writing the same code several times and
639     avoid the possibility of errors.
640
641                     SELECT DISTINCT CONCAT (
642                       UPPER(LEFT(`hello`, 1)),
643                       LOWER(SUBSTRING(`hello`, 2, LEN(`hello`) - 1))
644                       );
645
646     6.01. In the example below, we declare function `AP5.properUDP`.  We end
647           the name of the function with `UDP` to identify it as an user-defined
648           procedure.
649
650           The procedure has input parameter `@in_string` declared as
651           VARCHAR(50) -- in this case, the string `hello`.
652
653           We enclose the executable section between `BEGIN` and `END`.
654
655           We create output using parameter `@out_string`, which must have the
656           same file type as the input, in order to print (not return) the value
657           of `hello` as `Hello`.
658
659           Then we pass the value of `UPPER(LEFT(@in_string, 1)) +
660           LOWER(SUBSTRING(@in_string, 2, LEN(@in_string)-1))` to parameter
661           `@out_string`.
662  ************************************************************************ */
663
664 CREATE SCHEMA AP5;                              -- 1. creating schema `AP5` if
665                                                 --    not created already
666
667 CREATE PROCEDURE AP5.properUDP                  -- 2. creating stored procedure
668                                                 --    `AP5.properUDP`
669   @in_string VARCHAR(50)                        -- 3. declaring input parameter
670                                                 --    '@in_string` with data
671                                                 --    type `VARCHAR(50)`
```

```
AS
BEGIN                                           -- 4. beginning of executable
                                                --    code
  DECLARE @out_string VARCHAR(50)               -- 5. declaring output
                                                --    parameter `@out_string`
                                                --    with same data type as
                                                --    input parameter
                                                --    `@in_string` with data
                                                --    type `VARCHAR(50)`
  SET @out_string = CONCAT (                    -- 6. setting value of output
      UPPER(LEFT(@in_string, 1)),               --    parameter `@out_string`
      LOWER(SUBSTRING(@in_string, 2, LEN(@in_string) - 1))
      )
  PRINT @out_string;                            -- 7. printing (not returning)
                                                --    value of `@out_string`
END;                                            -- 8. end of executable code
                                                --    and stored procedure


/* ************************************************************************
    6.02. In order to execute (`EXEC`) our user-defined procedure (`UDP`), we
          must indicate the schema where it resides -- in this case, `AP5`.

          Since the output is printed (displayed only) and not returned, we
          cannot use the value by the procedure (`AP5.properUDP`).
 ************************************************************************ */

EXEC AP5.properUDP @in_string = 'hELLO';


/* ************************************************************************
    6.03. Procedures do not need input and/or output parameters if the
          executable code does not need parameters in order to work.

          In the example below, we have procedure `AP5.create_tempUDP` to
          create database `TEMP` and prints a message when it has been
          completed without the need of parameters.
 ************************************************************************ */

CREATE PROCEDURE AP5.create_tempUDP             -- 1. creating stored procedure
AS                                              --    `AP1.create_tempUDP`
BEGIN                                           -- 2. beginning of procedure
                                                --    executable code
  CREATE DATABASE TEMP                          -- 3. creating database `TEMP`
  PRINT 'Database Complete'                     -- 4. message displayed (not
                                                --    returned)
END;                                            -- 5. end of executable code
                                                --    and stored procedure

EXEC AP5.create_tempUDP;                        -- 6. executing procedure;
                                                --    no parameters needed in
                                                --    this example
```

```
724
725 /* ************************************************************************
726     6.04. In the example below, we have procedure `AP1.drop_tempUDP` to create
727           database `TEMP` and prints a message when it has been completed
728           without the need of parameters.
729  ************************************************************************ */
730
731 CREATE PROCEDURE AP5.drop_tempUDP              -- 1. creating stored procedure
732 AS                                             --    `AP1.drop_tempUDP`
733 BEGIN                                          -- 2. beginning of executable
734                                                --    code
735   DROP DATABASE TEMP                           -- 3. dropping database `TEMP`
736   PRINT 'Database Dropped'                     -- 4. message displayed (not
737                                                --    returned)
738 END;                                           -- 5. end of executable code
739                                                --    and stored procedure
740
741 EXEC AP5.drop_tempUDP;                         -- 6. executing procedure;
742                                                --    no parameters needed in
743                                                --    this example
744
745
746 /* ************************************************************************
747     6.05. As with all types of data objects, we can DROP procedures too.
748  ************************************************************************ */
749
750 DROP PROCEDURE AP5.create_tempUDP;             -- dropping procedure
751
752
753 /* ************************************************************************
754  7. In the two examples below, we have two procedures to change Celsius to
755     Fahrenheit and vice versa.
756
757     7.01. We first create schema `temps` in the `labs` database.
758  ************************************************************************ */
759
760 CREATE SCHEMA temps;
761
762
763 /* ************************************************************************
764     7.02. We create procedure `temps.c2f` taking in one parameter declared as a
765           FLOAT to convert temperatures in Celsius to Fahrenheit.
766  ************************************************************************ */
767
768 CREATE PROCEDURE temps.c2f @in_temp FLOAT      -- 1. input parameter
769                                                --    initialized as a FLOAT
770 AS
771 BEGIN
772   -- formula needed (9/5 C) + 32
773   DECLARE @out_temp FLOAT                      -- 2. declaring output
774                                                --    parameter `@out_temp`
775                                                --    with the same datatype as
```

```
776                                                 --    `@in_temp`, in this case
777                                                 --    a FLOAT
778   SET @out_temp = (9 / 5 * @in_temp) + 32       -- 3. formula to convert
779                                                 --    Celsius to Fahrenheit
780                                                 --    including `@in_temp`
781   DECLARE @out_result VARCHAR(150)              -- 4. new output to take the
782                                                 --    the value of
783   SET @out_result = CONCAT (                    -- 5. passing values including
784       CONVERT(VARCHAR(25), @in_temp),           --    `@in_temp` (temperature
785       'C = ',                                   --    in Celsius), `@out_temp`
786       CONVERT(VARCHAR(25), @out_temp),          --    `@out_temp` (temperature
787       'F'                                       --    in Fahrenheit)
788       )
789   PRINT @out_result                             -- 6. printing value to screen
790 END;
791
792 EXEC temps.c2f 75;                              -- 7. executing procedure
793                                                 --    `temps.c2f` passing 75
794                                                 --    as temperature in Celsius
795                                                 --    returning `75C = 107F`
796
797
798 /* ************************************************************************
799     7.03. We create procedure `temps.f2c` taking in one parameter declared as a
800           FLOAT to convert temperatures in Fahrenheit to Celsius.
801  ************************************************************************ */
802
803 CREATE PROCEDURE temps.f2c @in_temp FLOAT       -- 1. input parameter
804                                                 --    initialized as a FLOAT
805 AS
806 BEGIN
807   -- formula needed 5/9(F - 32)
808   DECLARE @out_temp FLOAT                       -- 2. declaring output
809                                                 --    parameter `@out_temp`
810                                                 --    with the same datatype as
811                                                 --    `@in_temp`, in this case
812                                                 --    a FLOAT
813   SET @out_temp = (@in_temp - 32) * 5 / 9       -- 3. formula to convert
814                                                 --    Fahrenheit to Celsius
815                                                 --    including `@in_temp`
816   DECLARE @out_result VARCHAR(150)              -- 4. new output to take the
817                                                 --    the value of
818   SET @out_result = CONCAT (                    -- 5. passing values including
819       CONVERT(VARCHAR(25), @in_temp),           --    `@in_temp` (temperature
820       'C = ',                                   --    in Fahrenheit),
821       CONVERT(VARCHAR(25), @out_temp),          --    `@out_temp` (temperature
822       'F'                                       --    in Celsius)
823       )
824   PRINT @out_result                             -- 6. printing value to screen
825 END;
826
827 EXEC temps.f2c 73;                              -- 7. executing procedure
```

```
828                                                 --    `temps.f2c` passing 73
829                                                 --    as temperature in
830                                                 --    Fahrenheit returning
831                                                 --    `73F = 22.7778C`
832
833
834  /* ************************************************************************
835   8. In SQL Server, a function is a stored program that we can pass parameters
836      into and return a value.
837      https://techonthenet.com/sql_server/functions.php
838
839                      CREATE FUNCTION function_name (@input_param data_type)
840                      RETURNS data_type
841                      AS
842                      BEGIN
843                        DECLARE @output_param data_type
844                        SET @output_param = some_value
845                        executable_code
846                        RETURN output_param
847                      END;
848
849      This also means that we can take the code that we used to capitalize the
850      first letter in a string and make it into a function that we can call
851      instead of writing the same code several times and avoid the possibility of
852      errors.
853
854                      SELECT DISTINCT CONCAT (
855                        UPPER(LEFT(`hello`, 1)),
856                        LOWER(SUBSTRING(`hello`, 2, LEN(`hello`) - 1))
857                        );
858
859      8.01. In the example below, we create function `AP5.properUDF`.  We end the
860            name of the function with `UDF` to identify it as an user-defined
861            function.  As explained before, no two objects of the same hierarchy
862            can have the same name.  Therefore our user-defined procedure and
863            function cannot share the name (`AP5.proper`) and a suffix tells
864            the system which object to use.
865
866            The function has input parameter `@in_string` declared as VARCHAR(50)
867            -- in this case, the string `hello`.
868
869            We enclose the executable section between `BEGIN` and `END`.
870
871            We create output using parameter `@out_string`, which must have the
872            same file type as the input parameter, in order to return the value of
873            `hello` as `Hello`.
874
875            Then we pass the value of concatenation
876
877                      CONCAT(
878                        UPPER(
879                          LEFT(@in_string, 1)
```

```
                        ),
                      LOWER(
                        SUBSTRING(@in_string, 2,
                          LEN(@in_string) - 1)
                          )

          or concatenation using the `+` sign

                    UPPER(
                      LEFT(@in_string, 1)
                      ) +
                     LOWER(
                       SUBSTRING(@in_string, 2,
                        LEN(@in_string) - 1)
                        )

          to parameter `@out_string`.

          As the last step, we must indicate what value must be returned from
          the function -- in this case, parameter `@out_string`.
 ********************************************************************** */

CREATE FUNCTION AP5.properUDF                   -- 1. creating function
                                                --    `AP5.properUDF`
  (@in_string VARCHAR(50))                      -- 2. declaring input parameter
                                                --    `@in_string` as
                                                --    VARCHAR(50)
RETURNS VARCHAR(50)                             -- 3. indicating the same data
                                                --    type and size of output
                                                --    parameter `@out_string`
AS
BEGIN                                           -- 4. beginning of executable
                                                --    code
  DECLARE @out_string VARCHAR(50)               -- 5. declaring output
                                                --    parameter `@out_string`
                                                --    with same data type as
                                                --    input parameter
                                                --    `@in_string`; same data
                                                --    type and size as
                                                --    indicated after `RETURNS`
  SET @out_string = CONCAT (                    -- 6. setting value to output
      UPPER(LEFT(@in_string, 1)),               --    parameter `@out_string`
      -- LOWER(SUBSTRING(@in_string, 2, LEN(@in_string) - 1))
      LOWER(RIGHT(@in_string, LEN(@in_string) - 1))
      )
  RETURN @out_string;                           -- 7. returning value of output
                                                --    parameter `@out_string`
END;                                            -- 8. end of executable code
                                                --    and function


/* **********************************************************************
```

```
932      8.02. In order to call our user-defined function (`UDF`), we must indicate
933            the schema where it resides -- in this case, `AP5`.
934   ************************************************************************** */
935
936   SELECT DISTINCT AP5.properUDF('hello');
937
938
939   /* ************************************************************************
940      8.03. We can use `AP5.properUDF` on any string value in any table, schema
941            or database as long as we have access to the data objects -- for
942            example, columns `AP2.Customers.FirstName` and
943            `AP2.Customers.LastName`.  Of course, first we insert values into
944            `AP2.Customers`.
945   ************************************************************************** */
946
947   INSERT INTO AP2.Customers                        -- all values in order
948   VALUES (
949     1,
950     'Smith',
951     'John',
952     '',
953     '',
954     '',
955     '',
956     ''
957     ),
958     (
959     2,
960     'Doe',
961     'Jane',
962     '123 Main St. Apt. 1',
963     'New York',
964     'NY',
965     '10001',
966     'jane.doe@foobar.foo'
967     );
968
969   INSERT INTO AP2.Customers (                      -- some values specifying the
970     CustomerID,                                    -- the order of the fields
971     LastName,
972     FirstName
973     )
974   VALUES (
975     3,
976     'Smith',
977     'Tom'
978     );
979
980   INSERT INTO AP2.Customers
981   VALUES (
982     5,
983     'Doe',
```

```
   'John',
   '',
   'New York',
   'NY',
   '10001',
   'john.doe@foobar.foo'
   ),
   (
   4,
   'Doe',
   'Jane',
   '',
   'New York',
   'NY',
   '',
   'jane.doe2@foobar.foo'
   );

UPDATE AP2.Customers
SET FirstName = AP5.properUDF(FirstName),
  LastName = AP5.properUDF(LastName);


/* ************************************************************************
    8.04. We can also create functions to FORMAT dollar amounts
          (`AP5.dollarUDF`) and dates (`AP5.dateUDF`) considering that numeric
          values become strings when formatted.
 ************************************************************************ */

CREATE FUNCTION AP5.dollarUDF (@in_dollar FLOAT)
RETURNS VARCHAR(50)
AS
BEGIN
  DECLARE @out_dollar VARCHAR(50)
  SET @out_dollar = FORMAT(@in_dollar, 'c', 'en-us')
  RETURN @out_dollar
END;

SELECT DISTINCT AP5.dollarUDF(10000) AS FormattedDollarAmout;

CREATE FUNCTION AP5.dateUDF (@in_date DATE)
RETURNS VARCHAR(10)
AS
BEGIN
  DECLARE @out_date VARCHAR(10)
  SET @out_date = FORMAT(@in_date, 'd', 'en-us')
  RETURN @out_date
END;

SELECT DISTINCT AP5.dateUDF(GETDATE()) AS FormattedDate;


```

```
1036 /* ************************************************************************
1037     8.05. Going back to procedures, we can call user-defined functions inside
1038           user-defined procedures (commonly referred to as stored procedures).
1039           We can call `AP5.properUDF` on `AP2.Customers.FirstName` and
1040           `AP2.Customers.LastName`.
1041  ************************************************************************ */
1042
1043 CREATE PROCEDURE AP5.properUDP                  -- 1. creating stored procedure
1044                                                 --    without input parameters
1045 AS
1046 BEGIN                                           -- 2. beginning of executable
1047                                                 --    code without parameters
1048   UPDATE AP2.Customers                          -- 3. updating `AP2.Customers`
1049   SET FirstName = AP5.properUDF(FirstName);
1050   PRINT 'Proper case assigned to first names';  -- 4. printing message
1051   UPDATE AP2.Customers                          -- 5. updating `AP2.Customers`
1052   SET LastName = AP5.properUDF(LastName);
1053   PRINT 'Proper case assigned to last names';   -- 6. printing message
1054 END;                                           -- 7. end of executable code
1055                                                 --    and stored procedure
1056
1057 CREATE PROCEDURE AP5.CloneInvoicesUDP           -- 1. creating stored procedure
1058                                                 --    `AP1.CloneInvoicesUDP`
1059 AS
1060 BEGIN                                           -- 2. beginning of executable
1061                                                 --    code
1062   DROP TABLE AP1.CloneInvoices;                 -- 3. dropping old clone table
1063   PRINT 'Old table `AP1.Invoices` destroyed';   -- 4. displaying completion
1064                                                 --    message
1065   SELECT DISTINCT                                  -- 5. SELECT
       DISTINCTing all values from
1066     InvoiceID,                                  --    `AP1.Invoices`
1067     VendorID,
1068     InvoiceNumber,
1069     AP5.dateUDF(InvoiceDate)                    -- 6. calling date values of
1070       AS InvoiceDate,                           --    columns using
1071     AP5.dollarUDF(InvoiceTotal)                 --    user-defined functions
1072       AS InvoiceTotal,                          --    `AP5.dateUDF` and
1073     AP5.dollarUDF(PaymentTotal)                 --    `AP5.dollarUDF`
1074       AS PaymentTotal,                          --
1075     AP5.dollarUDF(CreditTotal)
1076       AS CreditTotal,
1077     TermsID,
1078     AP5.dateUDF(InvoiceDueDate)
1079       AS InvoiceDueDate,
1080     AP5.dateUDF(PaymentDate)
1081       AS PaymentDate                            -- 7. pushing values from old
1082   INTO AP1.CloneInvoices                        --    table `AP1.Invoices` to
1083                                                 --    new table
1084                                                 --    `AP1.CloneInvoices`
1085   FROM AP1.Invoices;                            -- 8. from `AP1.Invoices`
1086   PRINT 'New table `AP5.Invoices` created';     -- 9. displaying completion
```

```
1087                                                    --    message
1088  END;                                              -- 10. end of executable code
1089                                                    --     and stored procedure
1090
1091  EXEC AP5.CloneInvoicesUDP;
1092
1093
1094  /* ********************************************************************************
1095   https://folvera.commons.gc.cuny.edu/?p=1237
1096   ******************************************************************************** */
```