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

  SESSION #7 (2021/11/23): CREATING DATABASE OBJECTS

  1. Understanding functions `CONVERT()`, `CAST()`, `DAY()`, `MONTH()`,
     `YEAR()` and `GETDATE()`
  2. Creating, dropping and altering views
 ************************************************************************

 1. As a quick review, SQL is the language to interact with a relational
    database.

    1.1. to request data (`SELECT`) from database objects like databases,
         schemas, tables and views
    1.2. to create (`CREATE`) where to store data, database objects like
         databases, schemas, tables including columns, etc.`
    1.3. to modify (`ALTER`) database objects
    1.4. to delete (`DROP`) database objects, automatic `COMMIT` in SQL Server
         hence no `ROLLBACK` (no way to rescue the data or objects)
    1.5. to manipulate data either affecting the data or not (showing data)

                    CREATE obj_type object_name
                    [other_code]

                    DROP obj_type object_name
                    [other_code]

                    ALTER obj_type object_name
                    ALTER|ADD|DROP obj_type obj_name data_type [other_code]

                    DELETE FROM table_name
                    [other_code]

                    INSERT INTO table_name
                    VALUES
                      (
                        field1 datatype1,
                        field2 datatype2
                        ...
                      )

                    TRUNCATE TABLE table_name

                    UPDATE table_name
                    SET field = new_value

    1.6. We use SQL to return data to any person or program that needs data.

```

```
53          1.6.1. Your boss or end user requests data as a report (`RPT`),
54                 graphic (`GIF`, `JPEG`, `PNG`, `BMP`, etc.), an office file
55                 (`XLS`, `DOC`, etc.) or other data types as `PDF`.  The end
56                 user does not need to know where the data comes from or how to
57                 get it.
58
59          1.6.2. You (the middle person handling errands) get the data from the
60                 database using SQL.  Normally you would not take care of
61                 visualization, analysis and/or interpretation.  You also do not
62                 need to understand the data, but you need to know your data
63                 (row ID of tables, keys or other constraints, etc.) and make
64                 sure the data is clean (no garbage data).
65
66          1.6.3. The database holds the data that the end user needs and all SQL
67                 requests (`SELECT`, `DROP`, `ALTER`, etc.) if written currently
68                 (no syntax errors) return the data to you or directly to the
69                 end user.  The database does not have AI and hence only returns
70                 what you ask.  As such if you make the wrong request, the
71                 database would return the wrong data (your bad logic, not a
72                 syntax error).
73  ***************************************************************************
74
75  2. In the example below, write a query without duplicate rows (`SELECT
76     DISTINCT`)
77     2.1. to call all shared values from tables `AP1.Invoices` and `AP1.Terms`,
78     2.2. to format all dates as `yyyy-MM-dd` and currency as pounds sterling
79          (culture `en-gb`)
80     2.3. where `AP1.Invoices.PaymentTotal` is greater than the average value of
81          `AP1.Invoices.InvoiceTotal` (sub-query within the `WHERE` clause) and
82          `AP1.Invoices.PaymentDate` is not null.
83  *************************************************************************** */
84
85  SELECT AP1.Invoices.InvoiceID,
86    AP1.Invoices.VendorID,
87    AP1.Invoices.InvoiceNumber,
88    FORMAT(AP1.Invoices.InvoiceDate,              -- 1. formatting column
89                                                  --    `InvoiceDate` as
90      'yyyy-MM-dd', 'en-gb')                      --    `yyyy-MM-dd` date with
91                                                  --    culture `en-gb` using
92       AS InvoiceDate,                            --    alias `InvoiceDate`
93    FORMAT(AP1.Invoices.InvoiceTotal,             -- 2. formatting column
94                                                  --    `InvoiceTotal` as
95      'c', 'en-gb')                               --    `c` (currency) with
96                                                  --    culture `en-gb` using
97      AS InvoiceTotal,                            --    alias `InvoiceTotal`
98    FORMAT(AP1.Invoices.PaymentTotal,             -- 3. formatting column
99      'c', 'en-gb')                               --    as `c` (currency) with
100                                                 --    culture `en-gb` using
101     AS PaymentTotal,                            --    alias `PaymentTotal`
102   FORMAT(AP1.Invoices.CreditTotal,              -- 4. formatting column
103     'c', 'en-gb')                               --    as `c` (currency) with
104                                                 --    culture `en-gb` using
```

```
105     AS CreditTotal,                             --    alias `CreditTotal`
106   AP1.Invoices.TermsID,
107   FORMAT(AP1.Invoices.InvoiceDueDate,           -- 5. formatting column
108     'yyyy-MM-dd', 'en-gb')                      --    as `yyyy-MM-dd` date with
109                                                 --    culture `en-gb` using
110     AS InvoiceDueDate,                          --    alias `InvoiceDueDate`
111   FORMAT(AP1.Invoices.PaymentDate,              -- 6. formatting column
112     'yyyy-MM-dd', 'en-gb')                      --    as `yyyy-MM-dd` date with
113                                                 --    culture `en-gb` using
114     AS PaymentDate,                             --    alias `PaymentDate`
115   AP1.Terms.TermsDescription,
116   AP1.Terms.TermsDueDays
117 FROM AP1.Invoices                              -- 7. from table `AP1.Invoices`
118 INNER JOIN AP1.Terms                           --    using `INNER JOIN` to
119                                                 --    retrieve all shared data
120   ON AP1.Invoices.TermsID = AP1.Terms.TermsID   --    connecting both tables on
121                                                 --    shared field `TermsID`
122 WHERE (                                        -- 8. where the value of
123     AP1.Invoices.PaymentTotal > (               --    `PaymentTotal` is larger
124                                                 --    than (`>`) the single
125       SELECT AVG(PaymentTotal)                  --    value of sub-query
126       FROM AP1.Invoices                         --        `(SELECT
127                                                 --         AVG(PaymentTotal)
128                                                 --         FROM AP1.Invoices)`
129                                                 --    that returns 1879.7413
130       )                                         --    8.1. sub-query always in
131     )                                           --         parenthesis, just
132                                                 --         like in algebra
133                                                 --    8.2. no need for
134                                                 --         `ORDER BY` since
135                                                 --         aggregate function
136                                                 --         `AVG()` affects only
137                                                 --         one column and it
138                                                 --         does not affect the
139                                                 --         main query
140   AND AP1.Invoices.PaymentDate IS NOT NULL;     -- 9. and [where] value of
141                                                 --    `PaymentDate` is not null
142                                                 --    (must have a value)
143
144
145 /* ****************************************************************************
146   3. Although using a custom format like `yyyy-MM-dd` overrides the culture
147      (`en-us`) and there is no longer need to include this culture, it is a
148      good idea to include it as good practice.
149  **************************************************************************** */
150
151 SELECT FORMAT(InvoiceTotal, 'yyyy-MM-dd')       -- no culture (`en-us`) needed
152 FROM AP1.Invoices;                              -- because of the custom format
153
154 SELECT FORMAT(InvoiceTotal,                     -- good practice to include the
155   'yyyy-MM-dd', 'en-us')                        -- culture (`en-us`) even when
156 FROM AP1.Invoices;                              -- overridden by custom format
```

```
/* ************************************************************************
  4. As mentioned several times, `FORMAT()` changes numeric values to strings.
     We can also use `CONVERT()` to change ``an expression from a data type to
     another data type`` -- in other words, numeric values to strings or vice
     versa (https://techonthenet.com/sql_server/functions/convert.php).

                    CONVERT(new_data_type, column)

     `CONVERT()` does not change the currency sign or adds commas to divide
     thousands or millions as `FORMAT()` does.

     4.1. In the example below, we change the data type of `InvoiceTotal` to
          VARCHAR(50) -- an allocation in RAM to hold a variable character
          value with a maximum size of fifty (50) characters.
 ************************************************************************ */

SELECT CONVERT(VARCHAR(50), InvoiceTotal)       -- changing data type of column
  AS InvoiceTotal                               -- `InvoiceTotal` (`FLOAT`) to
FROM AP1.Invoices;                              -- `VARCHAR(50)`


/* ************************************************************************
     4.2. In the example below, we use `CONVERT()` to return the value of
          `AP1.Invoices.InvoiceTotal` as a dollar amount concatenating the
          dollar sign (`$`) at the beginning.
 ************************************************************************ */

SELECT CONCAT (
        '$',                                    -- concatenating `$` to the
        CONVERT(VARCHAR(50), InvoiceTotal)      -- output of
        ) AS InvoiceTotal                       --    `CONVERT(VARCHAR(50),
FROM AP1.Invoices;                              --    `InvoiceTotal)`


/* ************************************************************************
          4.2.1. We could also use `CONVERT()` to return the value of
                 `AP1.Invoices.InvoiceTotal` as a dollar amount with `USD `
                 rather than the dollar sign (`$`).
 ************************************************************************ */

SELECT CONCAT (
        'USD ',                                 -- concatenating `USD ` to the
        CONVERT(VARCHAR(50), InvoiceTotal)      -- output of
        ) AS InvoiceTotal                       --    `CONVERT(VARCHAR(50),
FROM AP1.Invoices;                              --    `InvoiceTotal)`


/* ************************************************************************
          4.2.2. Of course, if you are ``dressing up`` a numeric value like
                 `AP1.Invoices.InvoiceTotal` as currency, it is better to just
                 use `FORMAT()` to keep your code simple.
 ************************************************************************ */
```

```
209
210 SELECT FORMAT(InvoiceTotal, 'c', 'en-us') AS InvoiceTotal
211 FROM AP1.Invoices;
212
213
214 /* ***************************************************************************
215      4.3. In the example below, we use `CONVERT()` to change the data type of
216           `AP1.Invoices.InvoiceID` and `AP1.Invoices.VendorID` from FLOAT to
217           `VARCHAR(50)` before concatenating these values to a string.
218  *************************************************************************** */
219
220 SELECT CONCAT (                                  -- 1. concatenating string
221                                                  --    values of
222     'Invoice ',                                  --    1.1. `Invoice `
223                                                  --         (hard-coded),
224     CONVERT(VARCHAR(3), InvoiceID),              --    1.2. the conversion of
225                                                  --         `InvoiceID` to
226                                                  --         `VARCHAR(3)`,
227     ' from vendor ',                             --    1.3. ` from vendor `
228                                                  --         (hard-coded) and
229     CONVERT(VARCHAR(3), VendorID)                --    1.4. the conversion of
230                                                  --         `VendorID` to
231                                                  --         `VARCHAR(3)`
232     ) AS InvoiceVendor                           --    1.5. using alias
233                                                  --         `InvoiceVendor`
234 FROM AP1.Invoices;
235
236
237 /* ***************************************************************************
238  5. We use the `WHERE` (https://techonthenet.com/sql_server/where.php)
239     clause to ``filter the results from a SELECT, INSERT, UPDATE, or DELETE
240     statement.``
241
242                     SELECT table1.field1, table1.field2 ...
243                       table2.field1, table2.field2 ...
244                     FROM table1
245                       INNER|LEFT|RIGHT JOIN table2
246                       ON table1.shared_field1 = table2.shared_field1
247                         AND table1.shared_field2 = table2.shared_field2
248                       ...
249                     WHERE condition1
250                       AND|OR condition2
251                       ...
252
253     5.1. We use conditions in order to filter data.
254
255          5.1.1. AND       to test for two or more conditions
256                           https://techonthenet.com/sql_server/and.php
257
258          5.1.2. OR        to test multiple conditions where records are
259                           returned when any one of the conditions are met
260                           https://techonthenet.com/sql_server/or.php
```

```
261
262     5.2. We use operators to compare values.
263
264          5.2.1. =         equal to
265                           https://techonthenet.com/sql_server/                ↵
                         comparison_operators.php
266
267          5.2.2. <>        not equal to
268                           https://techonthenet.com/sql_server/                ↵
                         comparison_operators.php
269
270          5.2.3. !=        not equal to
271                           https://techonthenet.com/sql_server/                ↵
                         comparison_operators.php
272
273          5.2.4. <         less than
274                           https://techonthenet.com/sql_server/                ↵
                         comparison_operators.php
275
276          5.2.5. >         greater than
277                           https://techonthenet.com/sql_server/                ↵
                         comparison_operators.php
278
279          5.2.6. <=        less than or equal to
280                           https://techonthenet.com/sql_server/                ↵
                         comparison_operators.php
281
282          5.2.7. >=        greater than or equal to
283                           https://techonthenet.com/sql_server/                ↵
                         comparison_operators.php
284
285          5.2.8. !>        not greater than (same as <=)
286                           https://techonthenet.com/sql_server/                ↵
                         comparison_operators.php
287
288          5.2.9. !<        not less than (same as >=)
289                           https://techonthenet.com/sql_server/                ↵
                         comparison_operators.php
290
291          5.2.10. LIKE     allows wild cards to be used in the WHERE clause of a
292                           SELECT, INSERT, UPDATE, or DELETE statement
293                           [allowing] you to perform pattern matching
294                           https://techonthenet.com/sql_server/like.php
295
296          5.2.11. IN       to help reduce the need to use multiple OR conditions
297                           in a SELECT, INSERT, UPDATE, or DELETE statement
298                           https://techonthenet.com/sql_server/in.php
299
300          5.2.12. BETWEEN  used to retrieve values within a range in a SELECT,
301                           INSERT, UPDATE, or DELETE statement
302                           https://techonthenet.com/sql_server/between.php
303
```

```
304           5.2.13. IS NULL  condition... used to test for a NULL no value
305                            https://techonthenet.com/sql_server/is_null.php
306
307           5.2.14. NOT      to negate a condition in a SELECT, INSERT, UPDATE, or
308                            DELETE statement
309                            https://techonthenet.com/sql_server/not.php
310
311                            * NOT LIKE
312                            * NOT IN
313                            * NOT BETWEEN
314                            * IS NOT NULL
315                              https://techonthenet.com/sql_server/is_not_null.php
316
317      5.3. In the example, below, we retrieve all values from table `AP1.Vendors`
318           where `VendorState` is equal to `CA` and `VendorCity` could either be
319           `Fresno` or `Sacramento`.
320
321           Use parenthesis for SQL (regardless of vendor/distribution) to process
322           the inner condition first
323
324                      (
325                        VendorCity = 'Fresno'
326                        OR VendorCity = 'Sacramento'
327                      )
328
329                  before the outer condition.
330   ************************************************************************** */
331
332   SELECT *
333   FROM AP1.Vendors
334   WHERE VendorState = 'CA'                          -- 1. inner criterion that must
335                                                     --    be true (satisfied)
336     AND (                                           -- 2. outer criterion that must
337                                                     --    be true composed of two
338                                                     --    sections where either
339                                                     --    could be true (satisfied)
340      VendorCity = 'Fresno'                          --    3.1. first criteria that
341                                                     --         could be met
342      OR VendorCity = 'Sacramento'                   --    3.2. second criteria that
343                                                     --         could be met
344      );
345
346
347   /* ************************************************************************
348      5.4. In the example below, we retrieve all values from table `AP1.Vendors`
349           where `VendorState` is not (`<>` or `!=`) `NY`.
350    ************************************************************************** */
351
352   SELECT *
353   FROM AP1.Vendors                                  -- can also be written as
354   WHERE VendorState <> 'NY';                        -- `VendorState != `NY``
355
```

```
356 /* ************************************************************************
357     5.5. In the example below, we retrieve all values from table `AP1.Vendors`
358          where `VendorState` is either `DC` or `IA`.
359  ************************************************************************ */
360
361 SELECT *
362 FROM AP1.Vendors
363 WHERE VendorState = 'DC'                        -- checking if either criterion
364   OR VendorState = 'IA';                        -- is true
365
366
367 /* ************************************************************************
368     5.6. In the example below, we retrieve all values from table `AP1.Vendors`
369          where `VendorAddress2` is NULL (no-value) using `NOT` as it negates
370          operators `LIKE` as `NOT LIKE`, `IN` as `NOT IN`, `BETWEEN` as
371          `NOT BETWEEN` and `IS NULL` as `IS NOT NULL`.
372  ************************************************************************ */
373
374 SELECT *
375 FROM AP1.Vendors
376 WHERE VendorAddress2 IS NULL;                   -- asking for no-value
377
378
379 /* ************************************************************************
380     5.7. In the example below, we retrieve all values from table `AP1.Vendors`
381          where `VendorAddress2` is not NULL (not a no-value).  Refer to
382          https://techonthenet.com/sql_server/is_not_null.php for more
383          information.
384  ************************************************************************ */
385
386 SELECT *
387 FROM AP1.Vendors
388 WHERE VendorAddress2 IS NOT NULL;               -- asking for not `NOT NULL`
389                                                 -- (no no-value)
390
391
392 /* ************************************************************************
393     5.8. In the example below, we rewrite #6.3 in a cleaner fashion to retrieve
394          all values from table `AP1.Vendors` where `VendorState` is equal to
395          `CA` and `VendorCity` could either be `Fresno` or `Sacramento`.  We
396          use operator `IN` (https://techonthenet.com/sql_server/in.php) to
397          specify the list of values that can be true for `VendorCity`.
398  ************************************************************************ */
399
400 SELECT *
401 FROM AP1.Vendors
402 WHERE VendorState = 'CA'                        -- 1. first condition as in
403                                                 --    original example
404   AND VendorCity IN (                           -- 2. second condition using
405     'Fresno',                                   --    `IN` to list all possible
406     'Sacramento'                                --    values that can be true
407     );                                          --    (satisfied)
```

```
408
409
410 /* ************************************************************************
411     5.9. In the example below, we retrieve all values from table `AP1.Vendors`
412          where `VendorState` could either be `CA` or `NJ` and `VendorCity`
413          could either be `Fresno` or `Sacramento`.
414
415          This query looks for the combination of
416
417                 `CA` and `Fresno`     (true)
418                 `CA` and `Sacramento` (true)
419
420          as well as
421
422                 `NJ` and `Fresno`     (false)
423                 `NJ` and `Sacramento` (false)
424
425          The query only returns only the first set of values since we do not
426          have any records where `VendorCity` is `NJ` and VendorCity` is either
427          `Fresno` or `Sacramento`.
428  ************************************************************************ */
429
430 SELECT *
431 FROM AP1.Vendors
432 WHERE (
433     VendorState IN (                          -- 1. first condition using
434       'CA',                                   --    `IN` to list all possible
435       'NJ'                                    --    values that can be true
436       )                                       --    (satisfied)
437     AND VendorCity IN (                       -- 2. second condition using
438       'Fresno',                               --    `IN` to list all possible
439       'Sacramento'                            --    values that can be true
440       )                                       --    (satisfied)
441     )
442 ORDER BY VendorState,
443   VendorCity;
444
445
446 /* ************************************************************************
447     5.10. In the example below, we retrieve all values from table `AP1.Vendors`
448           where `VendorState` could either be `CA` and `VendorCity` could
449           either be `Fresno` or `Sacramento` as one condition or `VendorState`
450           is `NJ` as another condition.
451  ************************************************************************ */
452
453 SELECT *
454 FROM AP1.Vendors
455 WHERE (                                       -- 1. first condition
456     VendorState IN ('CA')                     --    1.1. where `VendorState`
457                                               --         could be `CA`
458     AND VendorCity IN (                       --    1.2. and `VendorCity`
459       'Fresno',                               --         could either be
```

```
460         'Sacramento'                            --          `Fresno` or
461         )                                       --          `Sacramento`
462       )                                         --     looking the combinations
463                                                 --     of `CA` and `Fresno` or
464                                                 --     `CA` and `Sacramento`
465     OR VendorState IN ('NJ')                    -- 2. second condition starting
466                                                 --     with `OR` to specify that
467                                                 --     `VendorState` could also
468                                                 --     be `NJ`
469 ORDER BY VendorState,                          -- 3. ordering results first by
470   VendorCity;                                   --     `VendorState` and then by
471                                                 --     `VendorCity`
472
473
474 /* ********************************************************************************
475     5.11. In the example below, we retrieve all values from table `AP1.Vendors`
476           where `VendorName` has as a value starting with `am` (not case
477           sensitive) using wild card `%` to represent any character or group of
478           after `am`.
479  ******************************************************************************** */
480
481 SELECT *
482 FROM AP1.Vendors
483 WHERE VendorName LIKE 'am%';                    -- returns values
484                                                 -- `American Booksellers Assoc`
485                                                 -- and `American Express`
486
487
488 /* ********************************************************************************
489     5.12. In the example below, we retrieve all values from table `AP1.Vendors`
490           where `VendorName` has as a value with pattern `data` (not case
491           sensitive) using wild card `%` before and after the given string.
492  ******************************************************************************** */
493
494 SELECT *
495 FROM AP1.Vendors
496 WHERE VendorName LIKE '%data%';                 -- returns various values like
497                                                 -- `Expedata Inc`,
498                                                 -- `California Data Marketing`
499                                                 -- and `Quality Education Data`
500
501
502 /* ********************************************************************************
503     5.13. In the example below, we retrieve all values from table `AP1.Vendors`
504           where `VendorPhone` has as a value starting with `800` (string, not a
505           numeric value).
506  ******************************************************************************** */
507
508 SELECT *
509 FROM AP1.Vendors
510 WHERE VendorPhone LIKE '800%';
511
```

```
512
513 /* ************************************************************************
514     5.14. In the example below, we retrieve all values from table `AP1.Vendors`
515           where `VendorPhone` has as a value NOT starting with `800`.
516  ************************************************************************ */
517
518 SELECT *
519 FROM AP1.Vendors
520 WHERE VendorPhone NOT LIKE '800%';
521
522
523 /* ************************************************************************
524     5.15. In the example below, we retrieve all values from table
525           `AP1.Invoices` where `InvoiceDueDate` has values within the range of
526           two dates -- `2012-01-01` and `2012-01-30` (dates always in single
527           quotes).
528  ************************************************************************ */
529
530 SELECT *
531 FROM AP1.Invoices
532 WHERE InvoiceDueDate BETWEEN '2012-01-01'       -- range between `2012-01-01`
533     AND '2012-01-30';                           -- and `2012-01-30`
534
535
536 /* ************************************************************************
537     5.16. In the example below, we retrieve all values from table `AP1.Vendors`
538           where InvoiceTotal has values within 100 and 1000.  Then we organize
539           the results in descending order using an `ORDER BY` clause
540           (https://techonthenet.com/sql/order_by.php).
541
542           The default option for `ORDER BY` is `ASC` (ascending), which can be
543           omitted.
544
545           The opposite option for `ORDER BY` is `DESC` (descending), which
546           needs to be specified.
547  ************************************************************************ */
548
549 SELECT *
550 FROM AP1.Invoices
551 WHERE InvoiceTotal BETWEEN 100                  -- range between 100 and 1000
552     AND 1000
553 ORDER BY InvoiceTotal DESC,                     -- organizing results first by
554                                                 -- `InvoiceTotal` in descending
555                                                 -- order,
556   PaymentTotal DESC,                            -- then by `PaymentTotal` in
557                                                 -- descending order
558   TermsID DESC;                                 -- and finally by `TermsID`
559                                                 -- also in descending order
560
561
562 /* ************************************************************************
563  6. As we have mentioned several times, when calling multiple tables, we need
```

```
564      to `JOIN` them (https://techonthenet.com/sql_server/joins.php).
565
566      `INNER JOIN` returns ``all rows from multiple tables where the join
567      condition is met.``
568
569      6.1. In the example below, we retrieve all records shared in tables
570           `AP1.Invoices` and `AP1.Invoices`.
571   ********************************************************************** */
572
573  SELECT *
574  FROM AP1.Vendors
575  INNER JOIN AP1.Invoices
576    ON AP1.Vendors.VendorID = AP1.Invoices.VendorID;
577
578
579  /* ********************************************************************
580      6.2. `LEFT JOIN` returns ``all rows from the LEFT-hand table specified in
581           the ON condition and only those rows from the other table where the
582           joined fields are equal (join conditions met).``
583
584           6.2.1. In the example below, we retrieve all records in `AP1.Vendors`
585                  (left table) and any records in `AP1.Invoices` (if any in the
586                  right table).
587   ********************************************************************** */
588
589  SELECT *
590  FROM AP1.Vendors                                -- retrieves all records from
591  LEFT JOIN AP1.Invoices                          -- the left table/dataset
592    ON AP1.Vendors.VendorID = AP1.Invoices.VendorID; -- (first table/dataset
593                                                  -- called in the statement,
594                                                  -- `AP1.Vendors`) and related
595                                                  -- records from the right
596                                                  -- table/dataset (second
597                                                  -- table/dataset called in the
598                                                  -- statement, `AP1.Invoices`);
599                                                  -- returns 202
600
601
602  /* ********************************************************************
603           6.2.2. In the example below, we retrieve all records in `AP1.Invoices`
604                  (left table) and any records in `AP1.Vendors` (if any in the
605                  right table).
606   ********************************************************************** */
607
608  SELECT *
609  FROM AP1.Invoices                               -- retrieves all records from
610  LEFT JOIN AP1.Vendors                           -- the left table/dataset
611    ON AP1.Vendors.VendorID = AP1.Invoices.VendorID; -- (first table/dataset
612                                                  -- called in the statement,
613                                                  -- `AP1.Invoices`) and related
614                                                  -- records from the right
615                                                  -- table/dataset (second
```

```
                                                -- table/dataset called in the
                                                -- statement, `AP1.Vendors`)


/* ********************************************************************************
    6.3. `RIGHT JOIN` returns ``all rows from the RIGHT-hand table specified in
         the ON condition and only those rows from the other table where the
         joined fields are equal (join condition is met).``

         6.3.1. In the example below, we retrieve all records in `AP1.Invoices`
                (right table) and any records in `AP1.Vendors` (if any in the
                left table).
 ******************************************************************************** */

SELECT *
FROM AP1.Vendors                                -- retrieves all records from
RIGHT JOIN AP1.Invoices                         -- the right table/dataset
  ON AP1.Invoices.VendorID = AP1.Vendors.VendorID; -- (second table/dataset
                                                -- called in the statement,
                                                -- `AP1.Invoices`) and related
                                                -- records from the left
                                                -- table/dataset (first
                                                -- table/dataset called in the
                                                -- statement, `AP1.Invoices`)


/* ********************************************************************************
    6.4. `FULL JOIN` returns ``all rows from the LEFT-hand table and RIGHT hand
         table with nulls in place where the join condition is not met.``

         6.4.1. Depending on the size of the tables, this query might make the
                server run slowly or crash it.

         6.4.2. In the example below, we retrieve all records in `AP1.Invoices`
                (left table) and all records in `AP1.Vendors` (if any in the
                right table).
 ******************************************************************************** */

SELECT *
FROM AP1.Invoices                               -- retrieves all records from
FULL JOIN AP1.Vendors                           -- the left table/dataset
  ON AP1.Vendors.VendorID = AP1.Invoices.VendorID;  -- (first table/dataset
                                                -- called in the statement,
                                                -- `AP1.Vendors`) and all
                                                -- records from the right
                                                -- table/dataset (second
                                                -- table/dataset called in the
                                                -- statement, `AP1.Invoices`)


/* ********************************************************************************
 7. In the example below, we make some changes to `AP1.ContactUpdates` and
```

```
668      `AP1.Vendors`.
669
670      7.1. We add column `Email` to `AP1.ContactUpdates`, which should be
671           `VARCHAR(100)` and `NOT NULL` (HINT: `UPDATE` first, then `NOT NULL`).
672
673           7.1.1. First you need to add the column to the table.
674   ************************************************************************ */
675
676  ALTER TABLE AP1.ContactUpdates
677  ADD Email VARCHAR(100);
678
679
680  /* ************************************************************************
681      7.2. Then you have to populate the column (every field).
682
683           If you use `LastName` as part of the email, you should remove the
684           apostrophe in `O'Sullivan`.
685
686           Make sure to push the new values to an existant row in lower case
687           (HINT: `UPDATE`).
688   ************************************************************************ */
689
690  UPDATE AP1.ContactUpdates
691  SET Email = LOWER(CONCAT (
692        LEFT(FirstName, 1),                     -- 1. from `Geraldine`
693                                                --    returns `G`
694        REPLACE(LastName, '''', ''),            -- 2. from `O'Sullivan`
695                                                --    returns `OSullivan`
696        '@domain.web'                           -- 3. returns
697                                                --    `GOSullivan@domain.web`
698        ));                                     -- 4. returns
699                                                --    `gosullivan@domain.web`
700
701  /* ************************************************************************
702      7.3. Then you can change the column to `NOT NULL`.
703   ************************************************************************ */
704
705  ALTER TABLE AP1.ContactUpdates
706  ALTER COLUMN Email VARCHAR(100) NOT NULL;
707
708
709  /* ************************************************************************
710       7.4. We then add column `VendorAddress` to `AP1.Vendors`, which should be
711            `VARCHAR(150)` and `NOT NULL`.
712   ************************************************************************ */
713
714  ALTER TABLE AP1.Vendors
715  ADD VendorAddress VARCHAR(150);
716
717
718  /* ************************************************************************
719       7.5. Move the values of `VendorAddress1` and `VendorAddress2` to
```

```
720             `VendorAddress`.
721   ********************************************************************** */
722
723   UPDATE AP1.Vendors
724   SET VendorAddress = CONCAT (
725       VendorAddress1,
726       ' ',
727       VendorAddress2
728       );
729
730   /* **********************************************************************
731        7.6. Make sure the new column has the data and delete the original two
732             columns.
733   ********************************************************************** */
734
735   ALTER TABLE AP1.Vendors
736   DROP COLUMN VendorAddress1;
737
738   ALTER TABLE AP1.Vendors
739   DROP COLUMN VendorAddress2;
740
741
742   /* **********************************************************************
743        7.7. Change the new column to `NOT NULL`.
744   ********************************************************************** */
745
746   ALTER TABLE AP1.Vendors
747   ALTER COLUMN VendorAddress VARCHAR(150) NOT NULL;
748
749
750   /* **********************************************************************
751        7.8. Call all the values from `AP1.ContactUpdates` with any corresponding
752             values in `AP1.Vendors` (HINT: `LEFT JOIN` to get 8 records).
753   ********************************************************************** */
754
755   SELECT *
756   FROM AP1.ContactUpdates
757   LEFT JOIN AP1.Vendors
758     ON AP1.ContactUpdates.VendorID = AP1.Vendors.VendorID;
759
760
761   /* **********************************************************************
762        7.9. As a bonus, make a view named `AP1.ContactUpdates_VendorsVW` from the
763             prior query (#7.8).  See #9 for more information regarding views.
764   ********************************************************************** */
765
766   CREATE VIEW AP1.ContactUpdates_VendorsVW
767   AS
768   (
769       SELECT AP1.ContactUpdates.VendorID,
770         AP1.ContactUpdates.LastName,
771         AP1.ContactUpdates.FirstName,
```

```
772         AP1.ContactUpdates.Email,
773         -- AP1.Vendors.VendorID AS Expr1,
774         AP1.Vendors.VendorName,
775         AP1.Vendors.VendorCity,
776         AP1.Vendors.VendorState,
777         AP1.Vendors.VendorZipCode,
778         AP1.Vendors.VendorPhone,
779         AP1.Vendors.VendorContactLName,
780         AP1.Vendors.VendorContactFName,
781         AP1.Vendors.DefaultTermsID,
782         AP1.Vendors.DefaultAccountNo,
783         AP1.Vendors.VendorAddress
784       FROM AP1.ContactUpdates
785       LEFT JOIN AP1.Vendors
786         ON AP1.ContactUpdates.VendorID = AP1.Vendors.VendorID
787       );
788
789
790   /* ************************************************************************
791    8. Now that we have reviewed most of the material so far, we start views.
792
793           ``In a database management system, a view is a way of portraying
794           information in the database.  This can be done by arranging the data
795           items in a specific order, by highlighting certain items, or by
796           showing only certain items.  For any database, there are a number of
797           possible views that may be specified.  Databases with many items tend
798           to have more possible views than databases with few items.  Often
799           thought of as a virtual table, the view doesn't actually store
800           information itself, but just pulls it out of one or more existing
801           tables.  Although impermanent, a view may be accessed repeatedly by
802           storing its criteria in a query.``
803
804           http://searchsqlserver.techtarget.com/definition/view
805
806                       CREATE VIEW view_name AS
807                         SELECT columns
808                         FROM tables
809                         [WHERE conditions];
810
811       8.1. In the example below, we modify table `AP1.Invoices` adding column
812           `CustomerID` in order to establish a relation between this table and
813           `AP2.Customers`.
814    ************************************************************************ */
815
816   ALTER TABLE AP1.Invoices
817   ADD CustomerID INT NULL;
818
819   UPDATE AP1.Invoices
820   SET CustomerID = 1
821   WHERE VendorID = 34;
822
823   UPDATE AP1.Invoices
```

```
824  SET CustomerID = 2
825  WHERE VendorID = 37;
826
827  UPDATE AP1.Invoices
828  SET CustomerID = 3
829  WHERE VendorID = 89;
830
831
832  /* ***************************************************************************
833      8.2. Now that relationship has been created, we can now query tables
834           `AP1.Invoices` and `AP2.Customers` (each tables in a different
835           databases).
836   *************************************************************************** */
837
838  SELECT DISTINCT AP1.Invoices.InvoiceID,
839    AP1.Invoices.VendorID,
840    AP1.Invoices.InvoiceNumber,
841    FORMAT(AP1.Invoices.InvoiceDate, 'd', 'en-us') AS InvoiceDate,
842    FORMAT(AP1.Invoices.InvoiceTotal, 'c', 'en-us') AS InvoiceTotal,
843    FORMAT(AP1.Invoices.PaymentTotal, 'c', 'en-us') AS PaymentTotal,
844    FORMAT(AP1.Invoices.CreditTotal, 'c', 'en-us') AS CreditTotal,
845    AP1.Invoices.TermsID,
846    FORMAT(AP1.Invoices.InvoiceDueDate, 'd', 'en-us') AS InvoiceDueDate,
847    FORMAT(AP1.Invoices.PaymentDate, 'd', 'en-us') AS PaymentDate,
848    AP1.Invoices.CustomerID,
849    AP2.Customers.LastName,
850    AP2.Customers.FirstName,
851    AP2.Customers.Address,
852    AP2.Customers.City,
853    AP2.Customers.STATE,
854    AP2.Customers.ZipCode,
855    AP2.Customers.Email
856  FROM AP1.Invoices
857  INNER JOIN AP2.Customers
858    ON AP1.Invoices.CustomerID = AP2.Customers.CustomerID
859  ORDER BY AP1.Invoices.VendorID;
860
861
862  /* ***************************************************************************
863      8.3. In the example below, we can create a view using the query in the
864           example above using tables `AP1.Invoices` and `AP2.Customers` without
865           `ORDER BY`, which would return an error when creating the view.
866
867          Tables and views cannot share names since both data objects are of the
868          same hierarchy.
869
870          We can query, alter and/or drop a view just like a table.
871
872          In most relational databases, we cannot update data using a view since
873          this action only take place in tables.
874
875          In SQL Server (T-SQL), we can update data from the base table.
```

```
876
877                     ``Requires UPDATE, INSERT, or DELETE permissions on the
878                     target table, depending on the action being performed.``
879                     https://msdn.microsoft.com/en-us/library/ms180800.aspx
880  ********************************************************************** */
881
882 CREATE VIEW AP1.InvoicesCustomersVW
883 AS
884 (
885     SELECT DISTINCT AP1.Invoices.InvoiceID,
886       AP1.Invoices.VendorID,
887       AP1.Invoices.InvoiceNumber,
888       FORMAT(AP1.Invoices.InvoiceDate, 'd', 'en-us') AS InvoiceDate,
889       FORMAT(AP1.Invoices.InvoiceTotal, 'c', 'en-us') AS InvoiceTotal,
890       FORMAT(AP1.Invoices.PaymentTotal, 'c', 'en-us') AS PaymentTotal,
891       FORMAT(AP1.Invoices.CreditTotal, 'c', 'en-us') AS CreditTotal,
892       AP1.Invoices.TermsID,
893       FORMAT(AP1.Invoices.InvoiceDueDate, 'd', 'en-us') AS InvoiceDueDate,
894       FORMAT(AP1.Invoices.PaymentDate, 'd', 'en-us') AS PaymentDate,
895       AP1.Invoices.CustomerID,
896       AP2.Customers.LastName,
897       AP2.Customers.FirstName,
898       AP2.Customers.Address,
899       AP2.Customers.City,
900       AP2.Customers.STATE,
901       AP2.Customers.ZipCode,
902       AP2.Customers.Email
903     FROM AP1.Invoices
904     INNER JOIN AP2.Customers
905       ON AP1.Invoices.CustomerID = AP2.Customers.CustomerID
906     );
907
908
909 /* ************************************************************************
910     8.4. We can modify a view simply changing `CREATE` for `ALTER`.
911  ********************************************************************** */
912
913 ALTER VIEW AP1.InvoicesCustomersVW
914 AS
915 (
916     SELECT DISTINCT AP1.Invoices.InvoiceID,
917       AP1.Invoices.VendorID,
918       AP1.Invoices.InvoiceNumber,
919       FORMAT(AP1.Invoices.InvoiceDate, 'd', 'en-us')
920         AS InvoiceDate,
921       FORMAT(AP1.Invoices.InvoiceTotal, 'c', 'en-us') AS InvoiceTotal,
922       FORMAT(AP1.Invoices.PaymentTotal, 'c', 'en-us') AS PaymentTotal,
923       FORMAT(AP1.Invoices.CreditTotal, 'c', 'en-us') AS CreditTotal,
924       AP1.Invoices.TermsID,
925       FORMAT(AP1.Invoices.InvoiceDueDate, 'd', 'en-us') AS InvoiceDueDate,
926       FORMAT(AP1.Invoices.PaymentDate, 'd', 'en-us') AS PaymentDate,
927       AP1.Invoices.CustomerID,
```

```
928       AP2.Customers.LastName,
929       AP2.Customers.FirstName,
930       AP2.Customers.Address,
931       AP2.Customers.City,
932       AP2.Customers.STATE,
933       AP2.Customers.ZipCode,
934       AP2.Customers.Email,
935       GETDATE() AS SystemDate -- change in query
936     FROM AP1.Invoices
937     INNER JOIN AP2.Customers
938       ON AP1.Invoices.CustomerID = AP2.Customers.CustomerID
939     );
940
941
942  /* ***************************************************************************
943     8.5. In the example below, we create view `AP1.InvoicesVW` only from table
944          `AP1.Invoices` formatting the date and currency fields accordingly.
945          This way we do not need to format the columns again and again every
946          time we need to call them.
947   *************************************************************************** */
948
949  CREATE VIEW AP1.InvoicesVW
950  AS
951  (
952     SELECT DISTINCT InvoiceID,
953       VendorID,
954       InvoiceNumber,
955       FORMAT(InvoiceDate, 'd', 'en-us') AS InvoiceDate,
956       FORMAT(InvoiceTotal, 'c', 'en-us') AS InvoiceTotal,
957       FORMAT(PaymentTotal, 'c', 'en-us') AS PaymentTotal,
958       FORMAT(CreditTotal, 'c', 'en-us') AS CreditTotal,
959       TermsID,
960       FORMAT(InvoiceDueDate, 'd', 'en-us') AS InvoiceDueDate,
961       FORMAT(PaymentDate, 'd', 'en-us') AS PaymentDate,
962       CustomerID
963     FROM AP1.Invoices
964     );
965
966
967  /* ***************************************************************************
968     8.6. In the example below, we create view `AP1.InvoicesVendorsVW` from
969          tables `AP1.Invoices` and `AP1.Vendors`.
970
971          Unless we indicate in which database to store the view, it would most
972          likely be in the same database where the previous view was stored
973          (`AP2`).
974   *************************************************************************** */
975
976  CREATE VIEW AP1.InvoicesVendorsVW
977  AS
978  (
979     SELECT DISTINCT AP1.Invoices.InvoiceID,
```

```
980         AP1.Invoices.VendorID,
981         AP1.Invoices.InvoiceNumber,
982         AP1.Invoices.InvoiceDate,
983         AP1.Invoices.InvoiceTotal,
984         AP1.Invoices.PaymentTotal,
985         AP1.Invoices.CreditTotal,
986         AP1.Invoices.TermsID,
987         AP1.Invoices.InvoiceDueDate,
988         AP1.Invoices.PaymentDate,
989         AP1.Vendors.VendorName,
990         CASE
991           WHEN AP1.Vendors.VendorAddress2 IS NOT NULL
992             THEN CONCAT (
993                 AP1.Vendors.VendorAddress1,
994                 ' ',
995                 AP1.Vendors.VendorAddress2
996                 )
997           WHEN AP1.Vendors.VendorAddress1 IS NULL
998             AND AP1.Vendors.VendorAddress2 IS NULL
999             THEN 'No Address'
1000          ELSE AP1.Vendors.VendorAddress1
1001          END AS VendorAddress,
1002        AP1.Vendors.VendorCity,
1003        AP1.Vendors.VendorState,
1004        AP1.Vendors.VendorZipCode,
1005        AP1.Vendors.DefaultAccountNo
1006      FROM AP1.Invoices
1007      LEFT JOIN AP1.Vendors
1008        ON AP1.Invoices.VendorID = AP1.Vendors.VendorID
1009      );
1010
1011
1012  /* ***************************************************************************
1013      8.7. In the example below, we create view
1014           `AP1.Invoices_Customers_Vendors_VW` from views (like we would do with
1015           tables) `AP1.InvoicesCustomersVW` and `AP1.InvoicesVendorsVW`.
1016
1017           As mentioned, unless we indicate in which database to store the new
1018           view, it is saved in `AP2`.
1019
1020           We do not need to call the database and schema (`dbo`), but it is
1021           always a good idea -- good practice.
1022   ************************************************************************** */
1023
1024  CREATE VIEW AP1.Invoices_Customers_Vendors_VW
1025  AS
1026  (
1027      SELECT DISTINCT AP1.InvoicesCustomersVW.InvoiceID,
1028        AP1.InvoicesCustomersVW.VendorID,
1029        AP1.InvoicesCustomersVW.InvoiceNumber,
1030        AP1.InvoicesCustomersVW.InvoiceDate,
1031        AP1.InvoicesCustomersVW.InvoiceTotal,
```

```
1032        AP1.InvoicesCustomersVW.PaymentTotal,
1033        AP1.InvoicesCustomersVW.CreditTotal,
1034        AP1.InvoicesCustomersVW.TermsID,
1035        AP1.InvoicesCustomersVW.InvoiceDueDate,
1036        AP1.InvoicesCustomersVW.PaymentDate,
1037        AP1.InvoicesCustomersVW.CustomerID,
1038        AP1.InvoicesCustomersVW.LastName,
1039        AP1.InvoicesCustomersVW.FirstName,
1040        AP1.InvoicesCustomersVW.Address,
1041        AP1.InvoicesCustomersVW.City,
1042        AP1.InvoicesCustomersVW.STATE,
1043        AP1.InvoicesCustomersVW.ZipCode,
1044        AP1.InvoicesCustomersVW.Email,
1045        AP1.InvoicesVendorsVW.VendorName,
1046        AP1.InvoicesVendorsVW.VendorAddress,
1047        AP1.InvoicesVendorsVW.VendorCity,
1048        AP1.InvoicesVendorsVW.VendorState,
1049        AP1.InvoicesVendorsVW.VendorZipCode,
1050        AP1.InvoicesVendorsVW.DefaultAccountNo
1051      FROM AP1.InvoicesCustomersVW
1052      LEFT OUTER JOIN AP1.InvoicesVendorsVW
1053        ON AP1.InvoicesCustomersVW.VendorID = AP1.InvoicesVendorsVW.VendorID
1054      );
1055
1056
1057  /* ************************************************************************
1058   9. Depending on the relational database management system (RDBMS) and even the
1059      product related to each RDBMS, the date format might vary.  In SQL Server,
1060      we can query data using format `YYYY/MM/DD` (including quotes) although the
1061      system returns format `YYYY-MM-DD` plus time in format `hh:mm:ss.nnnnnnn`.
1062      Refer to https://msdn.microsoft.com/en-us/library/bb630352.aspx and
1063      https://msdn.microsoft.com/en-us/library/bb677243.aspx for information on
1064      date and time respectively.
1065
1066      9.1. The most common date functions are the following.
1067
1068           9.1.1. DAY       returns the day of the month (1 to 31) given a date
1069                            value
1070                            https://techonthenet.com/sql_server/functions/day.php
1071
1072           9.1.2. MONTH     returns the month (1 to 12) given a date value
1073                            https://techonthenet.com/sql_server/functions/
                         month.php
1074
1075           9.1.3. YEAR      returns a four-digit year (as a number) given a date
1076                            value
1077                            https://techonthenet.com/sql_server/functions/year.php
1078
1079           9.1.4. GETDATE   returns the current date and time
1080                            https://techonthenet.com/sql_server/functions/
                         getdate.php
1081    ************************************************************************ */
```

```
1082
1083  SELECT DAY('2021/09/15') AS Day,                -- 1. returns `15` from
1084                                                  --    `2021/09/15` without
1085                                                  --    leading zeros (`d`)
1086    MONTH('2021/09/15') AS Month,                 -- 2. returns `9` from
1087                                                  --    `2021/09/15` without
1088                                                  --    leading zeros (`M`)
1089    YEAR('2021/09/15') AS Year;                   -- 3. returns `2021` from
1090                                                  --    `2021/09/15 (`yyyy`)
1091
1092  SELECT GETDATE() AS CurrentDateTime;            -- returns
1093                                                  -- `2021-09-15 20:20:34.053`
1094                                                  -- from `GETDATE()` that calls
1095                                                  -- system date and time
1096
1097  SELECT DAY(GETDATE()) AS Day,                   -- 1. returns `15` from system
1098                                                  --    DATETIME without leading
1099                                                  --    zeros (`d`)
1100    MONTH(GETDATE()) AS Month,                    -- 2. returns `9` from system
1101                                                  --    DATETIME without leading
1102                                                  --    zeros (`M`)
1103    YEAR(GETDATE()) AS Year;                      -- 3. returns `2021` from
1104                                                  --    system DATETIME (`yyyy`)
1105
1106  SELECT FORMAT(GETDATE(), 'd', 'en-us')          -- returns system date and time
1107    AS FormattedCurrentDateTime;                  -- formatted as `9/15/2021`
1108
1109
1110  /* ************************************************************************
1111      9.2. Instead of hard-coding the date in the example above (#3.1), we can
1112           use parameter `@date` in all instances that we need to pass the value
1113           returned by `GETDATE()`.
1114
1115           We must declare each parameter with its proper data type.
1116
1117           We can then have to pass (`SET`) a value for each parameter.
1118   ************************************************************************ */
1119
1120  DECLARE @date DATETIME = GETDATE()              -- 1. declaring parameter
1121                                                  --    `@date` as DATETIME (the
1122                                                  --    proper data type) and
1123                                                  --    passing value of
1124                                                  --    `GETDATE()`
1125
1126  SELECT DAY(@date) AS Day,                       -- 2. returns `9` from system
1127                                                  --    DATETIME without leading
1128                                                  --    zeros (`d`)
1129    MONTH(@date) AS Month,                        -- 3. returns `15` from system
1130                                                  --    DATETIME without leading
1131                                                  --    zeros (`M`)
1132    YEAR(@date) AS Year;                          -- 4. returns `2021` from
1133                                                  --    system DATETIME (`yyyy`)
```

```
1134
1135
1136 /* ************************************************************************
1137     9.3. We can also use date function `GETDATE()` to calculate age in months,
1138          days and years.  The following script is based on the answer found at
1139          http://stackoverflow.com/q/57599/, which is explained below in detail.
1140
1141          9.3.1. We declare variables `@start_date`, `@end_date` and `@tmp_date`
1142                 as data type DATETIME
1143                 (https://msdn.microsoft.com/en-us/library/ms187819.aspx).
1144
1145          9.3.2. It is good practice to use a second variable (in this case,
1146                 `@tmp_date`) for calculations or other forms of data
1147                 manipulation.
1148
1149          9.3.3. We declare `@years`, `@months` and `@days` as INT
1150                 (https://msdn.microsoft.com/en-us/library/ms187745.aspx) for
1151                 date functions `DATEADD()` and `DATEDIFF()`.
1152  ************************************************************************ */
1153
1154 DECLARE @persons_name VARCHAR(100),             -- 1. person's first and last
1155                                                 --    names
1156   @start_date DATETIME,                         -- 2. person's birthday
1157   @end_date DATETIME,                           --    today's date from system
1158                                                 --    date and time
1159   @tmp_date DATETIME,                           -- 3. variable for calculations
1160   @years INT,                                   -- 4. variable for number of
1161                                                 --    years
1162   @months INT,                                  -- 5. variable for number of
1163                                                 --    months
1164   @days INT;                                    -- 6. variable for number of
1165                                                 --    days
1166
1167
1168 /* ************************************************************************
1169          9.3.4. We assign a value to `@start_date` as shown below since
1170                    there is no way for SQL Server to prompt the user to enter a
1171                    value.  In this example, we are using the date of birth of
1172                    Linus Torvalds (creator of the Linux kernel;
1173                    http://searchenterpriselinux.techtarget.com/definition/Linus-
                        Torvalds).
1174                    We also assign `GETDATE()` to `@end_date`.  This way we can
1175                    change the end date as needed (change from original query).
1176  ************************************************************************ */
1177
1178 SET @persons_name = 'Linus Torvalds',           -- person's name
1179   @start_date =     '12/28/1969',               -- person's date of birth
1180   @end_date =       GETDATE();                  -- today's system date and time
1181
1182
1183 /* ************************************************************************
1184          9.3.5. We assign the value of `@start_date` to `@tmp_date` to run
```

```
1185                    calculations against it and use `@start_date` as a constant.
1186   ************************************************************************** */
1187
1188  SELECT @tmp_date = @start_date;
1189
1190
1191  /* ************************************************************************
1192          9.3.6. Date functions `DATEADD()` returns ``a specified date with the
1193                 specified number interval (signed integer) added to a specified
1194                 datepart of that date``
1195                 (https://msdn.microsoft.com/en-us/library/ms186819.aspx) and
1196                 `DATEDIFF()` returns ``the count (signed integer) of the
1197                 specified datepart boundaries crossed between the specified
1198                 start_date and end_date``
1199                 (https://msdn.microsoft.com/en-us/library/ms189794.aspx).
1200
1201                `YEAR()` retrieves the year (`yy`) from the date.
1202
1203                `MONTH()` retrieves the month (`m`) from the date.
1204
1205                `DAY()` retrieves the day (`d`) from the date.
1206
1207          9.3.7. The `CASE WHEN` statement uses a true value (situation we are
1208                 looking for) coming from `WHEN... THEN` to trigger an action
1209                 and an `ELSE` value to trigger an alternative action using the
1210                 following syntax.
1211
1212          9.3.8. Below `@years` is assigned the difference of `@tmp_date` and
1213                 `@end_date` in years when the month in the year (`yy`) in
1214                 `@start_date` is less than the month in `@end_date` or it is
1215                 the same as the month in `@end_date`
1216
1217                     MONTH(@start_date) > MONTH(@end_date))
1218                       OR (MONTH(@start_date) = MONTH(@end_date))
1219
1220                 and the day in `@start_date` is less than the day in
1221                 `@end_date`.
1222
1223                     AND DAY(@start_date) > DAY(@end_date)
1224
1225                 If both conditions are true, the query returns `1` (under a
1226                 full year).  Otherwise it returns `0` (full year).
1227   ************************************************************************** */
1228
1229  SELECT @years = DATEDIFF(yy, @tmp_date, @end_date) - CASE
1230      WHEN (MONTH(@start_date) > MONTH(@end_date))
1231        OR (
1232          MONTH(@start_date) = MONTH(@end_date)
1233          AND DAY(@start_date) > DAY(@end_date)
1234          )
1235        THEN 1
1236      ELSE 0
```

```
1237      END;
1238
1239
1240  /* ************************************************************************
1241          9.3.9. We add the value of `@years` (`yy`) to `@tmp_date` returning 1
1242                 or 0.
1243   ************************************************************************ */
1244
1245  SELECT @tmp_date = DATEADD(yy, @years, @tmp_date);
1246
1247
1248  /* ************************************************************************
1249          9.3.10. Below `@months` is assigned the difference of `@tmp_date` and
1250                  `@end_date` in months when the month (`m`) in `@start_date` is
1251                  less than the month in `@end_date` or it is the same as the
1252                  month in `@end_date`.
1253
1254                     DAY(@start_date) > DAY(@end_date)
1255
1256                  If the condition is true, the query returns `1` (under a full
1257                  month).  Otherwise it returns `0` (full month).
1258   ************************************************************************ */
1259
1260  SELECT @months = DATEDIFF(m, @tmp_date, @end_date) - CASE
1261      WHEN DAY(@start_date) > DAY(@end_date)
1262        THEN 1
1263      ELSE 0
1264      END;
1265
1266
1267  /* ************************************************************************
1268          9.3.11. We add the value of `@months` (`m`) to `@tmp_date` returning 1
1269                  or 0.
1270   ************************************************************************ */
1271
1272  SELECT @tmp_date = DATEADD(m, @months, @tmp_date);
1273
1274
1275  /* ************************************************************************
1276          9.3.12. Below `@days` is assigned the difference of `@tmp_date` and
1277                  `@end_date` in days.
1278   ************************************************************************ */
1279
1280  SELECT @days = DATEDIFF(d, @tmp_date, @end_date);
1281
1282
1283  /* ************************************************************************
1284          9.3.13. We finally display the values for `@years`, `@months` and
1285                  `@days`.
1286
1287                     +---------------+-------+--------+------+
1288                     | Person's Name  | Years | Months | Days |
```

```
                    +---------------+-------+--------+------+
                    | Linus Torvalds | 51    | 10     | 26   |
                    +---------------+-------+--------+------+

          9.3.14. You can also use the script to calculate your age or any
                  difference between any two dates by changing the values in
                  section #9.3.4.

                  The value returned by `GETDATE()` when running this script was
                  2021/11/22 and the end result will change according to the
                  current date when the script is run.
 ********************************************************************** */

SELECT @persons_name AS 'Person''s Name',       -- two single quotes (``) to
                                                -- escape and show only one (`)
  @years AS 'Years',
  @months AS 'Months',
  @days AS 'Days';


/* ************************************************************************
10. LAB #6
    Write a query without duplicate rows (`SELECT DISTINCT`)
    10.1. to get all shared values from tables `AP1.InvoiceLineItems` and
          `AP1.GLAccounts` (`INNER JOIN`),
    10.2. adding today's date as `TodaysDate` formatted as short date
    10.3. 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
    10.4. 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))
    10.5. ordered first by `AP1.GLAccounts.AccountDescription` and then by
          `AP1.InvoiceLineItems.InvoiceLineItemAmount`.
 ********************************************************************** */

SELECT DISTINCT AP1.InvoiceLineItems.InvoiceID,
  AP1.InvoiceLineItems.InvoiceSequence,
  AP1.InvoiceLineItems.AccountNo,
  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
```

```
1341    (                                               -- 1. first block of two
1342                                                    --    conditions that must be
1343                                                    --    true
1344      AP1.GLAccounts.AccountDescription LIKE 'book%'
1345      AND AP1.InvoiceLineItems.InvoiceLineItemAmount >= 1000
1346      )
1347    OR                                              -- 2. `OR` to indicate that
1348                                                    --    either the first block
1349                                                    --    (above) or the second
1350                                                    --    (below) must be true
1351    (                                               -- 3. second block of two
1352                                                    --    conditions that must be
1353                                                    --    true
1354      AP1.GLAccounts.AccountDescription LIKE '%mail%'
1355      AND AP1.InvoiceLineItems.InvoiceLineItemAmount <= 100
1356      )
1357  ORDER BY AP1.GLAccounts.AccountDescription,
1358    AP1.InvoiceLineItems.InvoiceLineItemAmount,
1359    AP1.InvoiceLineItems.InvoiceID,
1360    AP1.InvoiceLineItems.InvoiceSequence,
1361    AP1.InvoiceLineItems.AccountNo,
1362    AP1.InvoiceLineItems.InvoiceLineItemDescription;
1363
1364
1365  /* ***************************************************************************
1366    11. LAB #7
1367        11.1. Create database `labs`.
1368        11.2. Create schema `lab7` in database `labs`.
1369        11.3. Create table `my_family` in schema `lab7` with the following
1370              structure choosing the best file type for each column and assign
1371              `NOT NULL` to each.
1372
1373                      row_id
1374                      person_fname
1375                      person_lname
1376                      relation
1377
1378        11.4. Insert values accordingly.
1379        11.5. Modify table `my_family` to add a column `dob`.
1380        11.6. Update the table with data in `dob` (new values in an existing
1381              record in table `labs.lab7.my_family`).
1382        11.7. Change column `dob` to `NOT NULL`.
1383   ************************************************************************** */
1384
1385  CREATE DATABASE labs;                             -- 1. creating database `labs`
1386                                                    --    1.1. run #1 (all `CREATE
1387                                                    --         DATABASE` statements
1388                                                    --         run together, but
1389                                                    --         separately from
1390                                                    --         other statements)
1391
1392  CREATE SCHEMA lab7;                               -- 2. creating schema `labs6`
```

```
                                                --    2.1. run #2 (each `CREATE
                                                --         SCHEMA` statement
                                                --         run separately)

CREATE TABLE lab7.my_family (                   -- 3. creating table
  row_id INT NOT NULL,                          --    `lab7.my_family`
  person_fname VARCHAR(25) NOT NULL,            --    3.1. run #3 (all `CREATE
  person_lname VARCHAR(25) NOT NULL,            --         TABLE` statement run
  relation VARCHAR(15) NOT NULL                 --         together, but
  );                                            --         separately from
                                                --         other statements)

INSERT INTO lab7.my_family                      -- 4. inserting new values into
VALUES (                                        --    table `lab7.my_family`
  1,                                            --    4.1. each row/record
  'John',                                       --         within a set of
  'Doe',                                        --         parenthesis followed
  'crazy uncle'                                 --         by a comma between
  ),                                            --         rows/records
  (                                             --    4.2. run #4 (all `INSERT`
  2,                                            --         statements run
  'Michael',                                    --         together, separately
  'Jones',                                      --         from other
  'cousin'                                      --         statements)
  ),
  (
  3,
  'Lucy',
  'Smith',
  'aunt'
  );

ALTER TABLE lab7.my_family                      -- 5. altering table
ADD dob DATE;                                   --    `lab7.my_family` to add
                                                --    column `dob` with data
                                                --    type `DATE`
                                                --    5.1. run #5 (all `ALTER`
                                                --         statements run
                                                --         together, separately
                                                --         from other
                                                --         statements)

UPDATE lab7.my_family                           -- 6. updating table
SET dob = '1970-01-01'                          --    `lab7.my_family` to pass
WHERE row_id = 1;                               --    a new values to column
                                                --    `dob` in the existing
UPDATE lab7.my_family                           --    table `lab7.my_family`
SET dob = '1980/05/09'                          --    6.1. run #6 (all `UPDATE`
WHERE row_id = 2;                               --         statements run
                                                --         together, separately
UPDATE lab7.my_family                           --         from other
SET dob = '1988/08/19'                          --         statements)
```

```
1445  WHERE row_id = 3;
1446
1447  ALTER TABLE lab7.my_family                    -- 7. changing new column `dob`
1448  ALTER COLUMN dob DATE NOT NULL;               --    to `NOT NULL` as column
1449                                                --    now has values
1450                                                --    7.1. run #7 (this `ALTER`
1451                                                --         statement run after
1452                                                --         populating new
1453                                                --         column `dob`
1454
1455
1456  /* ***************************************************************************
1457   https://folvera.commons.gc.cuny.edu/?p=1031
1458   ************************************************************************** */
```