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

 SESSION #7 (2021/11/29): 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
```

```
157
158
159 /* ***************************************************************************
160   4. As mentioned several times, `FORMAT()` changes numeric values to strings.
161      We can also use `CONVERT()` to change ``an expression from a data type to
162      another data type`` -- in other words, numeric values to strings or vice
163      versa (https://techonthenet.com/sql_server/functions/convert.php).
164
165                     CONVERT(new_data_type, column)
166
167      `CONVERT()` does not change the currency sign or adds commas to divide
168      thousands or millions as `FORMAT()` does.
169
170      4.1. In the example below, we change the data type of `InvoiceTotal` to
171           VARCHAR(50) -- an allocation in RAM to hold a variable character
172           value with a maximum size of fifty (50) characters.
173  *************************************************************************** */
174
175 SELECT CONVERT(VARCHAR(50), InvoiceTotal)       -- changing data type of column
176   AS InvoiceTotal                               -- `InvoiceTotal` (`FLOAT`) to
177 FROM AP1.Invoices;                              -- `VARCHAR(50)`
178
179
180 /* ***************************************************************************
181      4.2. In the example below, we use `CONVERT()` to return the value of
182           `AP1.Invoices.InvoiceTotal` as a dollar amount concatenating the
183           dollar sign (`$`) at the beginning.
184  *************************************************************************** */
185
186 SELECT CONCAT (
187         '$',                                    -- concatenating `$` to the
188         CONVERT(VARCHAR(50), InvoiceTotal)      -- output of
189         ) AS InvoiceTotal                       --    `CONVERT(VARCHAR(50),
190 FROM AP1.Invoices;                              --    `InvoiceTotal)`
191
192
193 /* ***************************************************************************
194           4.2.1. We could also use `CONVERT()` to return the value of
195                  `AP1.Invoices.InvoiceTotal` as a dollar amount with `USD `
196                  rather than the dollar sign (`$`).
197  *************************************************************************** */
198
199 SELECT CONCAT (
200         'USD ',                                 -- concatenating `USD ` to the
201         CONVERT(VARCHAR(50), InvoiceTotal)      -- output of
202         ) AS InvoiceTotal                       --    `CONVERT(VARCHAR(50),
203 FROM AP1.Invoices;                              --    `InvoiceTotal)`
204
205
206 /* ***************************************************************************
207           4.2.2. Of course, if you are ``dressing up`` a numeric value like
208                  `AP1.Invoices.InvoiceTotal` as currency, it is better to just
```

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

```
                          returned when any one of the conditions are met
                          https://techonthenet.com/sql_server/or.php

    5.2. We use operators to compare values.

         5.2.1. =         equal to
                          https://techonthenet.com/sql_server/
                     comparison_operators.php

         5.2.2. <>        not equal to
                          https://techonthenet.com/sql_server/
                     comparison_operators.php

         5.2.3. !=        not equal to
                          https://techonthenet.com/sql_server/
                     comparison_operators.php

         5.2.4. <         less than
                          https://techonthenet.com/sql_server/
                     comparison_operators.php

         5.2.5. >         greater than
                          https://techonthenet.com/sql_server/
                     comparison_operators.php

         5.2.6. <=        less than or equal to
                          https://techonthenet.com/sql_server/
                     comparison_operators.php

         5.2.7. >=        greater than or equal to
                          https://techonthenet.com/sql_server/
                     comparison_operators.php

         5.2.8. !>        not greater than (same as <=)
                          https://techonthenet.com/sql_server/
                     comparison_operators.php

         5.2.9. !<        not less than (same as >=)
                          https://techonthenet.com/sql_server/
                     comparison_operators.php

         5.2.10. LIKE     allows wild cards to be used in the WHERE clause of a
                          SELECT, INSERT, UPDATE, or DELETE statement
                          [allowing] you to perform pattern matching
                          https://techonthenet.com/sql_server/like.php

         5.2.11. IN       to help reduce the need to use multiple OR conditions
                          in a SELECT, INSERT, UPDATE, or DELETE statement
                          https://techonthenet.com/sql_server/in.php

         5.2.12. BETWEEN  used to retrieve values within a range in a SELECT,
                          INSERT, UPDATE, or DELETE statement
```

```
                          https://techonthenet.com/sql_server/between.php

         5.2.13. IS NULL  condition... used to test for a NULL no value
                          https://techonthenet.com/sql_server/is_null.php

         5.2.14. NOT      to negate a condition in a SELECT, INSERT, UPDATE, or
                          DELETE statement
                          https://techonthenet.com/sql_server/not.php

                          * NOT LIKE
                          * NOT IN
                          * NOT BETWEEN
                          * IS NOT NULL
                            https://techonthenet.com/sql_server/is_not_null.php

    5.3. In the example, below, we retrieve all values from table `AP1.Vendors`
         where `VendorState` is equal to `CA` and `VendorCity` could either be
         `Fresno` or `Sacramento`.

         Use parenthesis for SQL (regardless of vendor/distribution) to process
         the inner condition first

                    (
                      VendorCity = 'Fresno'
                      OR VendorCity = 'Sacramento'
                    )

         before the outer condition.
 ************************************************************************** */

SELECT *
FROM AP1.Vendors
WHERE VendorState = 'CA'                        -- 1. inner criterion that must
                                                --    be true (satisfied)
  AND (                                         -- 2. outer criterion that must
                                                --    be true composed of two
                                                --    sections where either
                                                --    could be true (satisfied)
    VendorCity = 'Fresno'                       --    3.1. first criteria that
                                                --         could be met
    OR VendorCity = 'Sacramento'                --    3.2. second criteria that
                                                --         could be met
    );


/* ************************************************************************
    5.4. In the example below, we retrieve all values from table `AP1.Vendors`
         where `VendorState` is not (`<>` or `!=`) `NY`.
 ************************************************************************** */

SELECT *
FROM AP1.Vendors                                -- can also be written as
```

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

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

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

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

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

```
                                                -- 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
    `AP1.Vendors`.

    7.1. We add column `Email` to `AP1.ContactUpdates`, which should be
         `VARCHAR(100)` and `NOT NULL` (HINT: `UPDATE` first, then `NOT NULL`).

         7.1.1. First you need to add the column to the table.
 ********************************************************************** */

ALTER TABLE AP1.ContactUpdates
ADD Email VARCHAR(100);


/* **********************************************************************
    7.2. Then you have to populate the column (every field).

         If you use `LastName` as part of the email, you should remove the
         apostrophe in `O'Sullivan`.

         Make sure to push the new values to an existant row in lower case
         (HINT: `UPDATE`).
 ********************************************************************** */

UPDATE AP1.ContactUpdates
SET Email = LOWER(CONCAT (
      LEFT(FirstName, 1),                    -- 1. from `Geraldine`
                                             --    returns `G`
      REPLACE(LastName, '''', ''),           -- 2. from `O'Sullivan`
                                             --    returns `OSullivan`
      '@domain.web'                          -- 3. returns
                                             --    `GOSullivan@domain.web`
      ));                                    -- 4. returns
                                             --    `gosullivan@domain.web`

/* **********************************************************************
    7.3. Then you can change the column to `NOT NULL`.
 ********************************************************************** */

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


/* **********************************************************************
     7.4. We then add column `VendorAddress` to `AP1.Vendors`, which should be
          `VARCHAR(150)` and `NOT NULL`.
 ********************************************************************** */

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


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

```
720       7.5. Move the values of `VendorAddress1` and `VendorAddress2` to
721            `VendorAddress`.
722    ********************************************************************** */
723
724  UPDATE AP1.Vendors
725  SET VendorAddress = CONCAT (
726      VendorAddress1,
727      ' ',
728      VendorAddress2
729      );
730
731  /* *********************************************************************
732       7.6. Make sure the new column has the data and delete the original two
733            columns.
734    ********************************************************************** */
735
736  ALTER TABLE AP1.Vendors
737  DROP COLUMN VendorAddress1;
738
739  ALTER TABLE AP1.Vendors
740  DROP COLUMN VendorAddress2;
741
742
743  /* *********************************************************************
744       7.7. Change the new column to `NOT NULL`.
745    ********************************************************************** */
746
747  ALTER TABLE AP1.Vendors
748  ALTER COLUMN VendorAddress VARCHAR(150) NOT NULL;
749
750
751  /* *********************************************************************
752       7.8. Call all the values from `AP1.ContactUpdates` with any corresponding
753            values in `AP1.Vendors` (HINT: `LEFT JOIN` to get 8 records).
754    ********************************************************************** */
755
756  SELECT *
757  FROM AP1.ContactUpdates
758  LEFT JOIN AP1.Vendors
759    ON AP1.ContactUpdates.VendorID = AP1.Vendors.VendorID;
760
761
762  /* *********************************************************************
763       7.9. As a bonus, make a view named `AP1.ContactUpdates_VendorsVW` from the
764            prior query (#7.8).  See #9 for more information regarding views.
765    ********************************************************************** */
766
767  CREATE VIEW AP1.ContactUpdates_VendorsVW
768  AS
769  (
770      SELECT AP1.ContactUpdates.VendorID,
771        AP1.ContactUpdates.LastName,
```

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

```sql
UPDATE AP1.Invoices
SET CustomerID = 2
WHERE VendorID = 37;

UPDATE AP1.Invoices
SET CustomerID = 3
WHERE VendorID = 89;


/* ***************************************************************************
    8.2. Now that relationship has been created, we can now query tables
         `AP1.Invoices` and `AP2.Customers` (each tables in a different
         databases).
 *************************************************************************** */

SELECT DISTINCT AP1.Invoices.InvoiceID,
  AP1.Invoices.VendorID,
  AP1.Invoices.InvoiceNumber,
  FORMAT(AP1.Invoices.InvoiceDate, 'd', 'en-us') AS InvoiceDate,
  FORMAT(AP1.Invoices.InvoiceTotal, 'c', 'en-us') AS InvoiceTotal,
  FORMAT(AP1.Invoices.PaymentTotal, 'c', 'en-us') AS PaymentTotal,
  FORMAT(AP1.Invoices.CreditTotal, 'c', 'en-us') AS CreditTotal,
  AP1.Invoices.TermsID,
  FORMAT(AP1.Invoices.InvoiceDueDate, 'd', 'en-us') AS InvoiceDueDate,
  FORMAT(AP1.Invoices.PaymentDate, 'd', 'en-us') AS PaymentDate,
  AP1.Invoices.CustomerID,
  AP2.Customers.LastName,
  AP2.Customers.FirstName,
  AP2.Customers.Address,
  AP2.Customers.City,
  AP2.Customers.STATE,
  AP2.Customers.ZipCode,
  AP2.Customers.Email
FROM AP1.Invoices
INNER JOIN AP2.Customers
  ON AP1.Invoices.CustomerID = AP2.Customers.CustomerID
ORDER BY AP1.Invoices.VendorID;


/* ***************************************************************************
    8.3. In the example below, we can create a view using the query in the
          example above using tables `AP1.Invoices` and `AP2.Customers` without
          `ORDER BY`, which would return an error when creating the view.

         Tables and views cannot share names since both data objects are of the
         same hierarchy.

         We can query, alter and/or drop a view just like a table.

         In most relational databases, we cannot update data using a view since
         this action only take place in tables.

```

```
876           In SQL Server (T-SQL), we can update data from the base table.
877
878                     ``Requires UPDATE, INSERT, or DELETE permissions on the
879                     target table, depending on the action being performed.``
880                     https://msdn.microsoft.com/en-us/library/ms180800.aspx
881   ************************************************************************** */
882
883  CREATE VIEW AP1.InvoicesCustomersVW
884  AS
885  (
886      SELECT DISTINCT AP1.Invoices.InvoiceID,
887        AP1.Invoices.VendorID,
888        AP1.Invoices.InvoiceNumber,
889        FORMAT(AP1.Invoices.InvoiceDate, 'd', 'en-us') AS InvoiceDate,
890        FORMAT(AP1.Invoices.InvoiceTotal, 'c', 'en-us') AS InvoiceTotal,
891        FORMAT(AP1.Invoices.PaymentTotal, 'c', 'en-us') AS PaymentTotal,
892        FORMAT(AP1.Invoices.CreditTotal, 'c', 'en-us') AS CreditTotal,
893        AP1.Invoices.TermsID,
894        FORMAT(AP1.Invoices.InvoiceDueDate, 'd', 'en-us') AS InvoiceDueDate,
895        FORMAT(AP1.Invoices.PaymentDate, 'd', 'en-us') AS PaymentDate,
896        AP1.Invoices.CustomerID,
897        AP2.Customers.LastName,
898        AP2.Customers.FirstName,
899        AP2.Customers.Address,
900        AP2.Customers.City,
901        AP2.Customers.STATE,
902        AP2.Customers.ZipCode,
903        AP2.Customers.Email
904      FROM AP1.Invoices
905      INNER JOIN AP2.Customers
906        ON AP1.Invoices.CustomerID = AP2.Customers.CustomerID
907      );
908
909
910  /* ************************************************************************
911      8.4. We can modify a view simply changing `CREATE` for `ALTER`.
912   ************************************************************************ */
913
914  ALTER VIEW AP1.InvoicesCustomersVW
915  AS
916  (
917      SELECT DISTINCT AP1.Invoices.InvoiceID,
918        AP1.Invoices.VendorID,
919        AP1.Invoices.InvoiceNumber,
920        FORMAT(AP1.Invoices.InvoiceDate, 'd', 'en-us')
921          AS InvoiceDate,
922        FORMAT(AP1.Invoices.InvoiceTotal, 'c', 'en-us') AS InvoiceTotal,
923        FORMAT(AP1.Invoices.PaymentTotal, 'c', 'en-us') AS PaymentTotal,
924        FORMAT(AP1.Invoices.CreditTotal, 'c', 'en-us') AS CreditTotal,
925        AP1.Invoices.TermsID,
926        FORMAT(AP1.Invoices.InvoiceDueDate, 'd', 'en-us') AS InvoiceDueDate,
927        FORMAT(AP1.Invoices.PaymentDate, 'd', 'en-us') AS PaymentDate,
```

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

```
    SELECT DISTINCT AP1.Invoices.InvoiceID,
      AP1.Invoices.VendorID,
      AP1.Invoices.InvoiceNumber,
      AP1.Invoices.InvoiceDate,
      AP1.Invoices.InvoiceTotal,
      AP1.Invoices.PaymentTotal,
      AP1.Invoices.CreditTotal,
      AP1.Invoices.TermsID,
      AP1.Invoices.InvoiceDueDate,
      AP1.Invoices.PaymentDate,
      AP1.Vendors.VendorName,
      CASE
        WHEN AP1.Vendors.VendorAddress2 IS NOT NULL
          THEN CONCAT (
              AP1.Vendors.VendorAddress1,
              ' ',
              AP1.Vendors.VendorAddress2
              )
        WHEN AP1.Vendors.VendorAddress1 IS NULL
          AND AP1.Vendors.VendorAddress2 IS NULL
          THEN 'No Address'
        ELSE AP1.Vendors.VendorAddress1
        END AS VendorAddress,
      AP1.Vendors.VendorCity,
      AP1.Vendors.VendorState,
      AP1.Vendors.VendorZipCode,
      AP1.Vendors.DefaultAccountNo
    FROM AP1.Invoices
    LEFT JOIN AP1.Vendors
      ON AP1.Invoices.VendorID = AP1.Vendors.VendorID
    );


/* ************************************************************************
    8.7. In the example below, we create view
         `AP1.Invoices_Customers_Vendors_VW` from views (like we would do with
         tables) `AP1.InvoicesCustomersVW` and `AP1.InvoicesVendorsVW`.

         As mentioned, unless we indicate in which database to store the new
         view, it is saved in `AP2`.

         We do not need to call the database and schema (`dbo`), but it is
         always a good idea -- good practice.
 ************************************************************************ */

CREATE VIEW AP1.Invoices_Customers_Vendors_VW
AS
(
    SELECT DISTINCT AP1.InvoicesCustomersVW.InvoiceID,
      AP1.InvoicesCustomersVW.VendorID,
      AP1.InvoicesCustomersVW.InvoiceNumber,
      AP1.InvoicesCustomersVW.InvoiceDate,
```

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

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

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

```
1185          9.3.5. We assign the value of `@start_date` to `@tmp_date` to run
1186                 calculations against it and use `@start_date` as a constant.
1187   ********************************************************************** */
1188
1189  SELECT @tmp_date = @start_date;
1190
1191
1192  /* **********************************************************************
1193          9.3.6. Date functions `DATEADD()` returns ``a specified date with the
1194                 specified number interval (signed integer) added to a specified
1195                 datepart of that date``
1196                 (https://msdn.microsoft.com/en-us/library/ms186819.aspx) and
1197                 `DATEDIFF()` returns ``the count (signed integer) of the
1198                 specified datepart boundaries crossed between the specified
1199                 start_date and end_date``
1200                 (https://msdn.microsoft.com/en-us/library/ms189794.aspx).
1201
1202                `YEAR()` retrieves the year (`yy`) from the date.
1203
1204                `MONTH()` retrieves the month (`m`) from the date.
1205
1206                `DAY()` retrieves the day (`d`) from the date.
1207
1208          9.3.7. The `CASE WHEN` statement uses a true value (situation we are
1209                 looking for) coming from `WHEN... THEN` to trigger an action
1210                 and an `ELSE` value to trigger an alternative action using the
1211                 following syntax.
1212
1213          9.3.8. Below `@years` is assigned the difference of `@tmp_date` and
1214                 `@end_date` in years when the month in the year (`yy`) in
1215                 `@start_date` is less than the month in `@end_date` or it is
1216                 the same as the month in `@end_date`
1217
1218                     MONTH(@start_date) > MONTH(@end_date))
1219                       OR (MONTH(@start_date) = MONTH(@end_date))
1220
1221                 and the day in `@start_date` is less than the day in
1222                 `@end_date`.
1223
1224                     AND DAY(@start_date) > DAY(@end_date)
1225
1226                 If both conditions are true, the query returns `1` (under a
1227                 full year).  Otherwise it returns `0` (full year).
1228   ********************************************************************** */
1229
1230  SELECT @years = DATEDIFF(yy, @tmp_date, @end_date) - CASE
1231      WHEN (MONTH(@start_date) > MONTH(@end_date))
1232        OR (
1233          MONTH(@start_date) = MONTH(@end_date)
1234          AND DAY(@start_date) > DAY(@end_date)
1235          )
1236        THEN 1
```

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

```
                    | Person's Name  | Years | Months | Days |
                    +----------------+-------+--------+------+
                    | Linus Torvalds | 51    | 11     | 2    |
                    +----------------+-------+--------+------+

          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
                  2021/11/29 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
```

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

```
1393  CREATE SCHEMA lab7;                          -- 2. creating schema `labs6`
1394                                               --    2.1. run #2 (each `CREATE
1395                                               --         SCHEMA` statement
1396                                               --         run separately)
1397
1398  CREATE TABLE lab7.my_family (                -- 3. creating table
1399    row_id INT NOT NULL,                       --    `lab7.my_family`
1400    person_fname VARCHAR(25) NOT NULL,         --    3.1. run #3 (all `CREATE
1401    person_lname VARCHAR(25) NOT NULL,         --         TABLE` statement run
1402    relation VARCHAR(15) NOT NULL              --         together, but
1403    );                                         --         separately from
1404                                               --         other statements)
1405
1406  INSERT INTO lab7.my_family                   -- 4. inserting new values into
1407  VALUES (                                     --    table `lab7.my_family`
1408    1,                                         --    4.1. each row/record
1409    'John',                                    --         within a set of
1410    'Doe',                                     --         parenthesis followed
1411    'crazy uncle'                              --         by a comma between
1412    ),                                         --         rows/records
1413    (                                          --    4.2. run #4 (all `INSERT`
1414    2,                                         --         statements run
1415    'Michael',                                 --         together, separately
1416    'Jones',                                   --         from other
1417    'cousin'                                   --         statements)
1418    ),
1419    (
1420    3,
1421    'Lucy',
1422    'Smith',
1423    'aunt'
1424    );
1425
1426  ALTER TABLE lab7.my_family                   -- 5. altering table
1427  ADD dob DATE;                                --    `lab7.my_family` to add
1428                                               --    column `dob` with data
1429                                               --    type `DATE`
1430                                               --    5.1. run #5 (all `ALTER`
1431                                               --         statements run
1432                                               --         together, separately
1433                                               --         from other
1434                                               --         statements)
1435
1436  UPDATE lab7.my_family                        -- 6. updating table
1437  SET dob = '1970-01-01'                       --    `lab7.my_family` to pass
1438  WHERE row_id = 1;                            --    a new values to column
1439                                               --    `dob` in the existing
1440  UPDATE lab7.my_family                        --    table `lab7.my_family`
1441  SET dob = '1980/05/09'                       --    6.1. run #6 (all `UPDATE`
1442  WHERE row_id = 2;                            --         statements run
1443                                               --         together, separately
1444  UPDATE lab7.my_family                        --         from other
```

```
1445  SET dob = '1988/08/19'                          --         statements)
1446  WHERE row_id = 3;
1447
1448  ALTER TABLE lab7.my_family                      -- 7. changing new column `dob`
1449  ALTER COLUMN dob DATE NOT NULL;                 --    to `NOT NULL` as column
1450                                                  --    now has values
1451                                                  --    7.1. run #7 (this `ALTER`
1452                                                  --         statement run after
1453                                                  --         populating new
1454                                                  --         column `dob`
1455
1456
1457  /* ***************************************************************************
1458   https://folvera.commons.gc.cuny.edu/?p=1037
1459   ************************************************************************** */
```