Steven Tautonico CST4714 Final Project Technical Documentation

CST4714 Database Administration Spring 2024
Prof G.

Table of contents

IMPORTANT NOTES

Most of the data generated and inserted into the database using python. This data could come from another source such as a CSV file or another database, but in this case, it was randomly generated using a python module called `Mimesis`.

All of this code is available on my GitHub (https://github.com/stautonico/CST4714-final)

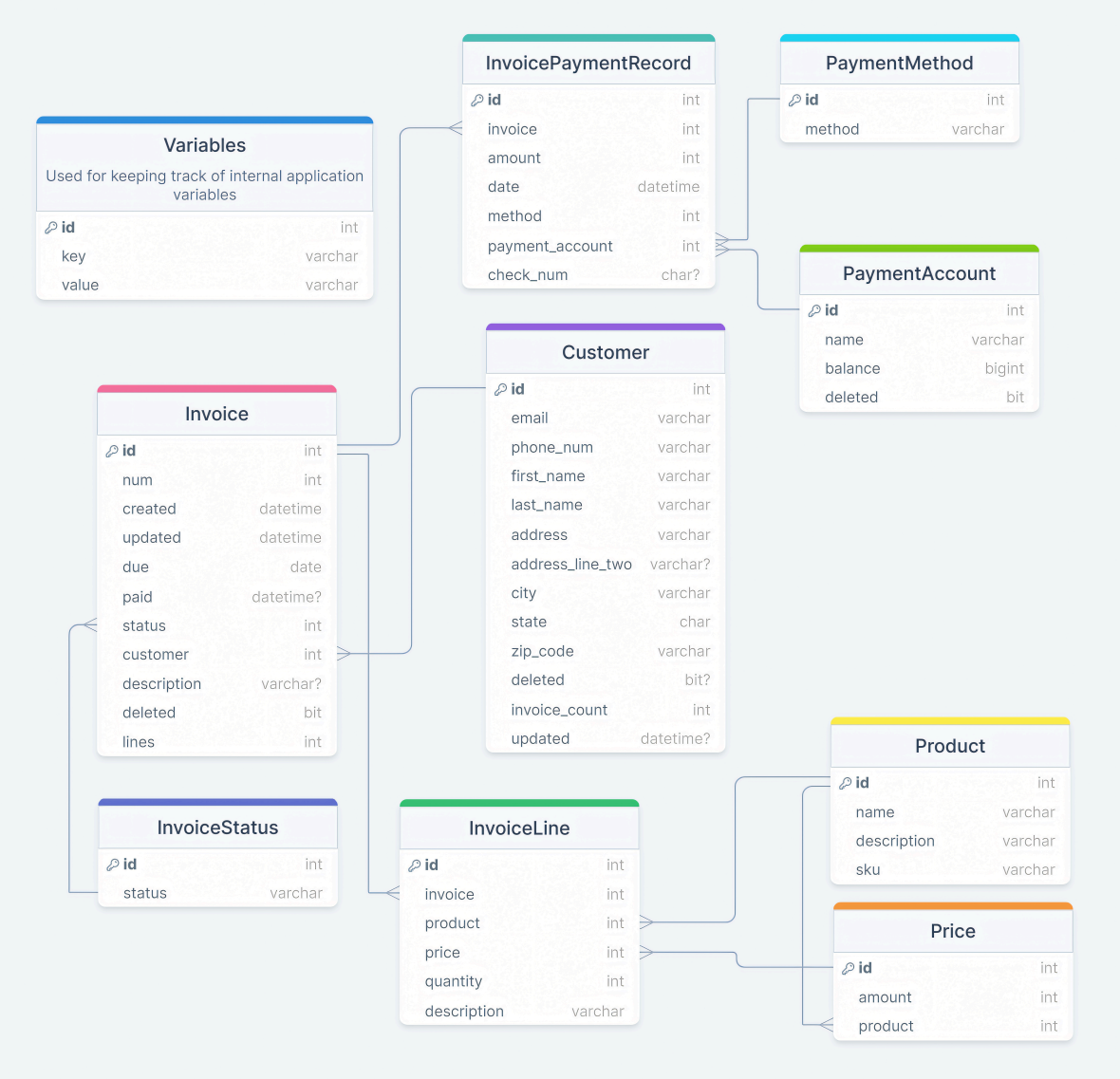
To run this code:

1. `cd python`
2. `pip install -r requirements.txt` (install dependencies)
3. `python3 reset_database.py` (drop the tables (if applicable) and re-create them (in the correct order))
4. `python3 generate_final_test_data.py` (this generates some fake data, inserts it into the database, then outputs a file called `payload.sql` will all the insert statements)

All the other files in the `python` folder were for development, and can be ignored

Deliverable 2

ERD



Erd

a. (Table code)

Invoice

```
CREATE TABLE S23916715.Invoice_ProfG_FP
(
```

```
    id          INT PRIMARY KEY IDENTITY (1,1),
    num         INT UNIQUE NOT NULL,
    created     DATETIME   NOT NULL DEFAULT SYSDATETIME(),
    updated     DATETIME   NOT NULL DEFAULT SYSDATETIME(),
    due         DATE       NOT NULL DEFAULT DATEADD(DAY, 30,
GETDATE()),
    paid        DATETIME, -- The timestamp when the invoice was
paid
    status      INT        NOT NULL,
    customer    INT        NOT NULL,
    description VARCHAR(1024),
    deleted     BIT        NOT NULL DEFAULT 0,
    lines       INT        NOT NULL DEFAULT 0,

FOREIGN KEY (customer) REFERENCES S23916715.Customer_ProfG_FP
(id),
FOREIGN KEY (status) REFERENCES S23916715.InvoiceStatus_ProfG_FP
(id)

)
```

Variables

```
-- Table used for keeping track of application variables
CREATE TABLE S23916715.Variables_ProfG_FP
(
    id    INT PRIMARY KEY IDENTITY (1,1),
    [key] VARCHAR(128) NOT NULL UNIQUE,
    value VARCHAR(128) NOT NULL
)
```

InvoicePaymentRecord

```
CREATE TABLE S23916715.InvoicePaymentRecord_ProfG_FP
(
    id              INT PRIMARY KEY IDENTITY (1,1),
    invoice         INT      NOT NULL,
```

```
    amount          INT      NOT NULL,
    date            DATETIME NOT NULL DEFAULT SYSDATETIME(),
    method          INT      NOT NULL,
    payment_account INT      NOT NULL,
    check_num       CHAR(8),

FOREIGN KEY (invoice) REFERENCES S23916715.Invoice_ProfG_FP (id),
FOREIGN KEY (method) REFERENCES S23916715.PaymentMethod_ProfG_FP
(id),
FOREIGN KEY (payment_account) REFERENCES
S23916715.PaymentAccount_ProfG_FP (id)

)
```

InvoiceLine

```
CREATE TABLE S23916715.InvoiceLine_ProfG_FP
(
    id          INT PRIMARY KEY IDENTITY (1,1),
    invoice     INT NOT NULL,
    product     INT NOT NULL,
    price       INT NOT NULL,
    quantity    INT NOT NULL,
    description VARCHAR(1024),

FOREIGN KEY (invoice) REFERENCES S23916715.Invoice_ProfG_FP (id),
FOREIGN KEY (product) REFERENCES S23916715.Product_ProfG_FP (id),
FOREIGN KEY (price) REFERENCES S23916715.Price_ProfG_FP (id),

)
```

PaymentMethod

```
CREATE TABLE S23916715.PaymentMethod_ProfG_FP
(
    id     INT PRIMARY KEY IDENTITY (1,1),
    method VARCHAR(20) NOT NULL, -- Possible values: CHECK, CASH,
```

```
CREDIT, WIRE, CRYPTO
)
```

Price

```
CREATE TABLE S23916715.Price_ProfG_FP
(
    id      INT PRIMARY KEY IDENTITY (1,1),
    amount  INT NOT NULL,
    product INT NOT NULL,

FOREIGN KEY (product) REFERENCES S23916715.Product_ProfG_FP (id)

)
```

Product

```
CREATE TABLE S23916715.Product_ProfG_FP
(
    id          INT PRIMARY KEY IDENTITY (1,1),
    name        VARCHAR(128) NOT NULL,
    description VARCHAR(1024),
    sku         VARCHAR(128) NOT NULL
)
```

Customer

```
CREATE TABLE S23916715.Customer_ProfG_FP
(
    id               INT PRIMARY KEY IDENTITY (1,1),
    email            VARCHAR(320) NOT NULL UNIQUE,
    phone_num        VARCHAR(15)  NOT NULL,
    first_name       VARCHAR(128) NOT NULL,
    last_name        VARCHAR(128) NOT NULL,
    address          VARCHAR(128) NOT NULL,
    address_line_two VARCHAR(128),
    city             VARCHAR(128) NOT NULL,
```

```
    state            CHAR(2)       NOT NULL,
    zip_code         VARCHAR(16)   NOT NULL,
    deleted          BIT                    DEFAULT 0,
    invoice_count    INT           NOT NULL DEFAULT 0,
    updated          DATETIME               DEFAULT SYSDATETIME()
)
```

InvoiceStatus

```
CREATE TABLE S23916715.InvoiceStatus_ProfG_FP
(
    id     INT PRIMARY KEY IDENTITY (1,1),
    status VARCHAR(10) NOT NULL,
)
```

PaymentAccount

```
CREATE TABLE S23916715.PaymentAccount_ProfG_FP
(
    id      INT PRIMARY KEY IDENTITY (1,1),
    name    VARCHAR(128) NOT NULL,
    balance INT          NOT NULL DEFAULT 0,
    deleted BIT          NOT NULL DEFAULT 0
)
```

Deliverable 4 (Reports)

1. Get Invoices by Customer

This stored procedure takes either a customer's email address or the customer id number and returns all invoices that are associated with the given customer

```
CREATE OR ALTER PROCEDURE
S23916715.GetInvoicesByCustomer_ProfG_FP(
    @customer_email VARCHAR(320) = NULL,
    @customer_id INT = NULL
)
AS
BEGIN
    SET NOCOUNT ON
    -- We need either the email or the id, not both, not neither
    IF @customer_email IS NULL AND @customer_id IS NULL
        BEGIN
            PRINT 'Please provide either the customer email or the
customer id';
            RETURN
        END

IF @customer_email IS NOT NULL AND @customer_id IS NOT NULL
    BEGIN
        PRINT 'Please provide either the customer email or the
customer id but not both';
        RETURN
    END

-- Check if the customer exists
DECLARE @findCustomerCount INT;
SELECT @findCustomerCount = COUNT(*)
FROM S23916715.Customer_ProfG_FP
WHERE (@customer_email IS NOT NULL AND email = @customer_email)
   OR (@customer_id IS NOT NULL AND id = @customer_id);
```

```
IF @findCustomerCount = 0
    BEGIN
        PRINT 'Customer does not exist'
        RETURN
    END


SELECT i.id                          AS InvoiceID,
       i.num                         AS InvoiceNumber,
       i.created                     AS InvoiceCreated,
       i.updated                     AS InvoiceUpdated,
       i.status                      AS InvoiceStatus,
       i.description                 AS InvoiceDescription,
       c.id                          AS CustomerID,
       maskedCustomer.masked_email AS CustomerEmail,
       c.first_name                  AS CustomerFirstName,
       c.last_name                   AS CustomerLastName
FROM S23916715.Invoice_ProfG_FP AS i
         INNER JOIN
     S23916715.Customer_ProfG_FP AS c ON i.customer = c.id
         INNER JOIN View_Customers_ProfG_FP AS maskedCustomer ON
i.customer = maskedCustomer.id
WHERE (@customer_email IS NOT NULL AND c.email = @customer_email)
   OR (@customer_id IS NOT NULL AND c.id = @customer_id)
AND i.deleted = 0;

SET NOCOUNT OFF

END
```

2. Get Invoices from this Month

This stored procedure gets all the invoices that were created in the current month. It has an optional argument to specify an invoice status to filer by

```
CREATE OR ALTER PROCEDURE
S23916715.GetInvoicesFromThisMonth_ProfG_FP(
    @status VARCHAR(10) = NULL
)
AS
BEGIN
    SET NOCOUNT ON

-- Select all invoices that have a creation date that is within
the last 30 days
-- Optionally, include the status if it was provided

SELECT i.id          AS InvoiceID,
       i.num         AS InvoiceNumber,
       i.created     AS InvoiceCreated,
       i.updated     AS InvoiceUpdated,
       i.status      AS InvoiceStatus,
       i.description AS InvoiceDescription,
       c.id          AS CustomerID,
       c.email       AS CustomerEmail,
       c.first_name  AS CustomerFirstName,
       c.last_name   AS CustomerLastName
FROM S23916715.Invoice_ProfG_FP AS i
         INNER JOIN
     S23916715.Customer_ProfG_FP AS c ON i.customer = c.id
WHERE YEAR(created) = YEAR(GETDATE())
  AND MONTH(created) = MONTH(GETDATE())
  -- I'm pretty sure this server is configured to be case-
insensitive,
  -- so we don't need to worry about changing the case (the db
should store all upper case)
  AND (@status IS NULL OR status =
S23916715.GetInvoiceStatusId_ProfF_FP(@status))
  AND i.deleted = 0;

SET NOCOUNT OFF
```

```
END
```

3. Customers that have Unpaid Invoices

This view returns all customers that have an invoice which isn't paid or canceled. This view implements data masking to hide sensitive fields such as phone number and email addresses.

```
CREATE OR ALTER VIEW
S23916715.CustomersThatHaveUnpaidInvoices_ProfG_FP AS
SELECT c.id,
       c.masked_email,
       c.masked_phone_number,
       c.first_name,
       c.last_name,
       COUNT(i.id) AS unpaid_invoice_count
FROM S23916715.View_Customers_ProfG_FP c
         LEFT JOIN S23916715.Invoice_ProfG_FP i ON c.id =
i.customer
WHERE i.status NOT IN
(S23916715.GetInvoiceStatusId_ProfF_FP('PAID'),
S23916715.GetInvoiceStatusId_ProfF_FP('DRAFT'),

S23916715.GetInvoiceStatusId_ProfF_FP('CANCELED'))
   OR i.id IS NULL AND i.deleted = 0
GROUP BY c.id, c.masked_email, c.masked_phone_number,
c.first_name, c.last_name;
```

4. Customer View

This view displays all the customers registered in the system. This view implements data masking to hide sensitive fields such as phone number and email addresses.

```
CREATE OR ALTER VIEW S23916715.View_Customers_ProfG_FP AS
SELECT id,
       CONCAT(
```

```
                LEFT(email, 1), -- Grab the first letter from the
username
                REPLICATE('*',
                          LEN(SUBSTRING(email, 2, CHARINDEX('@',
email) - 2))), -- Mask everything in between the first and last
letter of the username portion of the address
                RIGHT(email, 1), -- Grab the first letter from the
username
                SUBSTRING(email, CHARINDEX('@', email), LEN(email))
-- Append the domain
        ) AS masked_email,
        CONCAT(
                '***-***-',
                SUBSTRING(phone_num, 9, 4)
        ) AS masked_phone_number,
        first_name,
        last_name,
        address,
        address_line_two,
        city,
        state,
        zip_code
FROM S23916715.Customer_ProfG_FP
WHERE deleted = 0;
```

5. Get Unpaid Invoices

This stored procedure gets all invoices that aren't `PAID`, `DRAFT` OR `CANCELED`. This procedure also has an optional `asJson` argument which will output the result as a JSON string. This procedure implements data masking that disguises customer email and phone number.

```
CREATE OR ALTER PROCEDURE S23916715.GetUnpaidInvoices(
    @asJson BIT = 0
)
AS
BEGIN
```

```
    SET NOCOUNT ON;

CREATE TABLE #UnpaidInvoices
(
    InvoiceID          INT,
    InvoiceNumber      INT,
    InvoiceCreated     DATETIME,
    InvoiceUpdated     DATETIME,
    InvoiceStatus      VARCHAR(10),
    InvoiceDescription VARCHAR(1024),
    CustomerID         INT,
    CustomerEmail      VARCHAR(320),
    CustomerFirstName  VARCHAR(128),
    CustomerLastName   VARCHAR(128)
)

INSERT INTO #UnpaidInvoices (InvoiceID, InvoiceNumber,
InvoiceCreated, InvoiceUpdated, InvoiceStatus,
                             InvoiceDescription, CustomerID,
CustomerEmail, CustomerFirstName, CustomerLastName)
SELECT i.id                         AS InvoiceID,
       i.num                        AS InvoiceNumber,
       i.created                    AS InvoiceCreated,
       i.updated                    AS InvoiceUpdated,
       i.status                     AS InvoiceStatus,
       i.description                AS InvoiceDescription,
       c.id                         AS CustomerID,
       maskedCustomer.masked_email AS CustomerEmail,
       c.first_name                 AS CustomerFirstName,
       c.last_name                  AS CustomerLastName
FROM S23916715.Invoice_ProfG_FP AS i
         INNER JOIN
     S23916715.Customer_ProfG_FP AS c ON i.customer = c.id
         INNER JOIN View_Customers_ProfG_FP AS maskedCustomer ON
i.customer = maskedCustomer.id
WHERE i.status NOT IN (S23916715.GetInvoiceStatusId('PAID'),
S23916715.GetInvoiceStatusId('DRAFT'),
                       S23916715.GetInvoiceStatusId('CANCELED'))
```

```
  AND i.deleted = 0;

IF @asJson = 1
    BEGIN
        -- This is what happens when we're not allowed to use *
        SELECT InvoiceID,
               InvoiceNumber,
               InvoiceCreated,
               InvoiceUpdated,
               InvoiceStatus,
               InvoiceDescription,
               CustomerID,
               CustomerEmail,
               CustomerFirstName,
               CustomerLastName
        FROM #UnpaidInvoices
        FOR JSON PATH;
    END
ELSE
    BEGIN
        -- This is what happens when we're not allowed to use *
        SELECT InvoiceID,
               InvoiceNumber,
               InvoiceCreated,
               InvoiceUpdated,
               InvoiceStatus,
               InvoiceDescription,
               CustomerID,
               CustomerEmail,
               CustomerFirstName,
               CustomerLastName
        FROM #UnpaidInvoices;
    END

DROP TABLE #UnpaidInvoices;

SET NOCOUNT OFF
```

```
END
```

6. Get Paid Invoices

This stored procedure gets all invoices that have a status of `PAID` This procedure also has an optional `asJson` argument which will output the result as a JSON string. This procedure implements data masking that disguises customer email and phone number.

```
CREATE OR ALTER PROCEDURE S23916715.GetPaidInvoices(
    @asJson BIT = 0
)
AS
BEGIN
    SET NOCOUNT ON;

CREATE TABLE #PaidInvoices
(
    InvoiceID          INT,
    InvoiceNumber      INT,
    InvoiceCreated     DATETIME,
    InvoiceUpdated     DATETIME,
    InvoiceStatus      VARCHAR(10),
    InvoiceDescription VARCHAR(1024),
    CustomerID         INT,
    CustomerEmail      VARCHAR(320),
    CustomerFirstName  VARCHAR(128),
    CustomerLastName   VARCHAR(128)
)

INSERT INTO #PaidInvoices (InvoiceID, InvoiceNumber,
InvoiceCreated, InvoiceUpdated, InvoiceStatus,
                           InvoiceDescription, CustomerID,
CustomerEmail, CustomerFirstName, CustomerLastName)
SELECT i.id                          AS InvoiceID,
       i.num                         AS InvoiceNumber,
       i.created                     AS InvoiceCreated,
```

```
       i.updated                    AS InvoiceUpdated,
       i.status                     AS InvoiceStatus,
       i.description                AS InvoiceDescription,
       c.id                         AS CustomerID,
       maskedCustomer.masked_email AS CustomerEmail,
       c.first_name                 AS CustomerFirstName,
       c.last_name                  AS CustomerLastName
FROM S23916715.Invoice_ProfG_FP AS i
         INNER JOIN
     S23916715.Customer_ProfG_FP AS c ON i.customer = c.id
         INNER JOIN View_Customers_ProfG_FP AS maskedCustomer ON
i.customer = maskedCustomer.id
WHERE i.status = S23916715.GetInvoiceStatusId_ProfF_FP('PAID')
AND i.deleted = 0;

IF @asJson = 1
    BEGIN
        -- This is what happens when we're not allowed to use * :(
        SELECT InvoiceID,
               InvoiceNumber,
               InvoiceCreated,
               InvoiceUpdated,
               InvoiceStatus,
               InvoiceDescription,
               CustomerID,
               CustomerEmail,
               CustomerFirstName,
               CustomerLastName
        FROM #PaidInvoices
        FOR JSON PATH;
    END
ELSE
    BEGIN
        -- This is what happens when we're not allowed to use * :(
        SELECT InvoiceID,
               InvoiceNumber,
               InvoiceCreated,
               InvoiceUpdated,
```

```
                InvoiceStatus,
                InvoiceDescription,
                CustomerID,
                CustomerEmail,
                CustomerFirstName,
                CustomerLastName
        FROM #PaidInvoices;
    END

DROP TABLE #PaidInvoices;

SET NOCOUNT OFF

END
```

7. Get Stale Invoices

This view returns all invoices that are "stale." A stale invoice is defined as an invoice that hasn't been updated in at least 30 days. This view implements data masking that disguises customer email and phone number.

```
-- Get invoices that haven't been modified in at least 30 days
CREATE OR ALTER VIEW S23916715.GetStaleInvoices_ProfG_FP AS
SELECT i.id                          AS InvoiceID,
       i.num                         AS InvoiceNumber,
       i.created                     AS InvoiceCreated,
       i.updated                     AS InvoiceUpdated,
       i.status                      AS InvoiceStatus,
       i.description                 AS InvoiceDescription,
       c.id                          AS CustomerID,
       maskedCustomer.masked_email AS CustomerEmail,
       c.first_name                  AS CustomerFirstName,
       c.last_name                   AS CustomerLastName
FROM S23916715.Invoice_ProfG_FP AS i
         INNER JOIN
     S23916715.Customer_ProfG_FP AS c ON i.customer = c.id
         INNER JOIN View_Customers_ProfG_FP AS maskedCustomer ON
```

```
i.customer = maskedCustomer.id
-- In theory, this also checks if the date is >= 30 days in the
future, but the updated field
-- should never be in the future so it doesn't really matter
WHERE DATEDIFF(DAY, i.updated, GETDATE()) >= 30
AND i.deleted = 0;
```

8. Get Invoices by Number

This procedure gets an invoice provided the invoice number (different from the id.) This procedure has an optional `asJson` argument which will output the result as a JSON string. This procedure implements data masking that disguises customer email and phone number.

```
CREATE OR ALTER PROCEDURE S23916715.GetInvoiceByNum_ProfG_FP(
    @invoice_num INT,
    @asJson BIT = 0
)
AS
BEGIN
    SET NOCOUNT ON

CREATE TABLE #invoices
(
    InvoiceID          INT,
    InvoiceNumber      INT,
    InvoiceCreated     DATETIME,
    InvoiceUpdated     DATETIME,
    InvoiceStatus      VARCHAR(10),
    InvoiceDescription VARCHAR(1024),
    CustomerEmail      VARCHAR(320),
    CustomerFirstName  VARCHAR(128),
    CustomerLastName   VARCHAR(128),
    Total              FLOAT
);

INSERT INTO #invoices (InvoiceID, InvoiceNumber, InvoiceCreated,
```

```
InvoiceUpdated, InvoiceStatus, InvoiceDescription,
                        CustomerEmail, CustomerFirstName,
CustomerLastName, Total)
SELECT i.id                                                    AS
InvoiceID,
       i.num                                                   AS
InvoiceNumber,
       i.created                                               AS
InvoiceCreated,
       i.updated                                               AS
InvoiceUpdated,
       i.status                                                AS
InvoiceStatus,
       i.description                                           AS
InvoiceDescription,
       maskedCustomer.masked_email                             AS
CustomerEmail,
       c.first_name                                            AS
CustomerFirstName,
       c.last_name                                             AS
CustomerLastName,
       -- We store all money amounts as ints (aka x100) since
computers make floating point mistakes.
       -- When we want to display a money amount, we have to
divide by 100
       TRY_CAST(SUM(il.quantity * p.amount) AS FLOAT) / 100 AS
Total
FROM S23916715.Invoice_ProfG_FP AS i
         INNER JOIN S23916715.Customer_ProfG_FP AS c ON i.customer
= c.id
         INNER JOIN View_Customers_ProfG_FP AS maskedCustomer ON
i.customer = maskedCustomer.id
         INNER JOIN S23916715.InvoiceLine_ProfG_FP AS il ON i.id =
il.invoice
         INNER JOIN S23916715.Price_ProfG_FP AS p ON il.price =
p.id
WHERE i.num = @invoice_num
GROUP BY i.id, i.num, i.created, i.updated, i.status,
```

```
i.description, maskedCustomer.masked_email,
         c.first_name, c.last_name;

IF @asJson = 1
    BEGIN
        -- This is what happens when we're not allowed to use * >:
(
        SELECT InvoiceID,
               InvoiceNumber,
               InvoiceCreated,
               InvoiceUpdated,
               InvoiceStatus,
               InvoiceDescription,
               CustomerEmail,
               CustomerFirstName,
               CustomerLastName,
               Total
        FROM #invoices
        FOR JSON PATH;
    END
ELSE
    BEGIN
        -- This is what happens when we're not allowed to use * >:
(
        SELECT InvoiceID,
               InvoiceNumber,
               InvoiceCreated,
               InvoiceUpdated,
               InvoiceStatus,
               InvoiceDescription,
               CustomerEmail,
               CustomerFirstName,
               CustomerLastName,
               Total
        FROM #invoices;
    END

DROP TABLE #invoices;
```

```
SET NOCOUNT OFF

END
```

9. Get Prices for Product

This procedure gets all the prices associated with a product. This procedure takes either a product id or a product name & sku. If both or neither are provided, the procedure throws an error

```
CREATE OR ALTER PROCEDURE S23916715.GetPricesForProduct_ProfG_FP(
    @product_id INT = NULL,
    @product_name VARCHAR(128) = NULL,
    @product_sku VARCHAR(128) = NULL
)
AS
BEGIN
    -- We can't have both id and product&sku
    IF @product_id IS NOT NULL AND @product_sku IS NOT NULL AND
@product_name IS NOT NULL
        BEGIN
            PRINT 'You can''t provide both product ID and product
name&sku'
            RETURN
        END
        -- We have to have at least one of the two
    ELSE
        IF @product_id IS NULL AND (@product_name IS NULL OR
@product_sku IS NULL)
            BEGIN
                PRINT 'You must provide at least product ID or
product name&sku'
                RETURN
            END
```

```
-- If we don't have the product id, we need to find it
IF @product_id IS NULL
    BEGIN
        SELECT @product_id = id FROM S23916715.Product_ProfG_FP
WHERE name = @product_name AND sku = @product_sku;

        IF @product_sku IS NULL
            BEGIN
                PRINT CONCAT('Product with name ', @product_name,
' and sku ', @product_sku, ' doesn''t exist')
                RETURN
            END
    END

SELECT id,
       TRY_CAST(amount AS FLOAT) / 100 AS price
FROM S23916715.Price_ProfG_FP
WHERE product = @product_id;

END
```

10. Get Total Billed this Month

This view gets all invoices that were created during the current month, aren't canceled and aren't marked as deleted, then totals up the amount billed. The value is returned as a float.

```
CREATE OR ALTER VIEW S23916715.GetTotalBilledThisMonth_ProfG_FP AS
SELECT CONVERT(FLOAT, SUM(CONVERT(BIGINT, Total))) / 100 AS
GrandTotal
FROM (SELECT SUM(CONVERT(BIGINT, il.quantity * p.amount)) AS Total
      FROM S23916715.Invoice_ProfG_FP AS i
               INNER JOIN S23916715.InvoiceLine_ProfG_FP AS il ON
i.id = il.invoice
               INNER JOIN S23916715.Price_ProfG_FP AS p ON
il.price = p.id
      WHERE YEAR(created) = YEAR(GETDATE())
```

```
        AND MONTH(created) = MONTH(GETDATE())
        AND status != 'CANCELLED'
      AND deleted = 0
) AS Invoices;
```

11. Get Total Paid this Month

This view gets all payment records for invoices that were created during the current month, then totals up the amount billed. The value is returned as a float. This view also includes payments on invoices that aren't 100% paid off (partial payments.)

```
CREATE OR ALTER VIEW S23916715.GetTotalPaidThisMonth_ProfG_FP AS
SELECT CONVERT(FLOAT, SUM(CONVERT(BIGINT, Total))) / 100 AS
GrandTotal
FROM (SELECT SUM(CONVERT(BIGINT, amount)) AS Total
      FROM S23916715.InvoicePaymentRecord_ProfG_FP
      WHERE YEAR(date) = YEAR(GETDATE())
        AND MONTH(date) = MONTH(GETDATE())
) AS Records;
```

12. Calculate Remaining Balance

This function, given an invoice number, calculates the remaining balance on an invoice by totaling up the amount paid on every payment record that refers to the given invoice and subtracting the total cost. This function is used in several places for business logic but can also be used to generate reporting data.

```
CREATE OR ALTER FUNCTION
S23916715.CalculateRemainingBalance_ProfF_FP(@invoice_num INT)
    RETURNS FLOAT
AS
BEGIN
    -- Try and find the invoice id
    DECLARE @invoice_id INT;

SELECT @invoice_id = id FROM S23916715.Invoice_ProfG_FP WHERE num
= @invoice_num;
```

```
IF @invoice_id IS NULL
    RETURN -1

-- Go grab the invoice and its total cost
DECLARE @totalValue BIGINT;

SELECT @totalValue = SUM(il.quantity * p.amount)
FROM S23916715.Invoice_ProfG_FP AS i
         INNER JOIN S23916715.InvoiceLine_ProfG_FP AS il ON i.id =
il.invoice
         INNER JOIN S23916715.Price_ProfG_FP AS p ON il.price =
p.id
WHERE i.num = @invoice_num;

-- Now total up the amount paid
DECLARE @amountPaid BIGINT;

SET @amountPaid = S23916715.FindAmountPaid_ProfF_FP(@invoice_num);

RETURN TRY_CAST(@totalValue - @amountPaid AS FLOAT) / 100;

END
```

13. Get Invoice Payments

This procedure gets all payments associated with the given invoice number. This procedure implements data masking by only displaying the last four digits of the check number (if one exists)

```
CREATE OR ALTER PROCEDURE
S23916715.View_GetInvoicePayments_ProfG_FP(
    @invoice_num INT
)
AS
BEGIN
    SET NOCOUNT ON
```

```
-- TODO: Mask the check number if there is one
-- TODO: THIS SHIT DON'T WORK. I think its something wrong with
the joins. Break it down and try later

-- Step 1: Make sure our invoice exists
DECLARE @invoice_id INT;

SELECT @invoice_id = id FROM S23916715.Invoice_ProfG_FP WHERE num
= @invoice_num;

IF @invoice_id IS NULL
    BEGIN
        PRINT CONCAT('Invoice with num ', @invoice_num, ' doesn''t
exist')
        RETURN
    END

-- Step 2: Grab all the invoice payment records that match our
invoice's id,
--         as well as their associated methods and invoice
SELECT ipr.id,
       i.num,
       ipr.amount,
       ipr.date,
       m.method,
       IIF(ipr.check_num IS NOT NULL, CONCAT(REPLICATE('*', 4),
RIGHT(ipr.check_num, 4)), NULL) AS maksed_check_num
FROM S23916715.InvoicePaymentRecord_ProfG_FP AS ipr
         INNER JOIN S23916715.Invoice_ProfG_FP i ON i.id =
ipr.invoice
         INNER JOIN S23916715.PaymentMethod_ProfG_FP m ON m.id =
ipr.method
WHERE ipr.invoice = @invoice_id;

SET NOCOUNT OFF
```

```
END
```

14. Find Amount Paid

This function, given an invoice number, calculates the balance paid on an invoice by totaling up the amount paid on every payment record that refers to the given invoice. This function is used in several places for business logic but can also be used to generate reporting data.

```
CREATE OR ALTER FUNCTION
S23916715.FindAmountPaid_ProfF_FP(@invoice_num INT)
    RETURNS FLOAT
AS
BEGIN
    -- Try and find the invoice id
    DECLARE @invoice_id INT;

SELECT @invoice_id = id FROM S23916715.Invoice_ProfG_FP WHERE num
= @invoice_num;

IF @invoice_id IS NULL
    RETURN -1

-- Find all of the payment records that reference this invoice
DECLARE @amountPaid BIGINT;

SELECT @amountPaid = SUM(amount)
FROM S23916715.InvoicePaymentRecord_ProfG_FP
WHERE invoice = @invoice_id;

RETURN @amountPaid;

END
```

15. Get Total Billed this Year

This view gets all invoices that were created during the current year, aren't canceled and aren't marked as deleted, then totals up the amount billed. The value is returned as a float.

```
CREATE OR ALTER VIEW S23916715.GetTotalBilledThisYear_ProfG_FP AS
SELECT CONVERT(FLOAT, SUM(CONVERT(BIGINT, Total))) / 100 AS
GrandTotal
FROM (SELECT SUM(CONVERT(BIGINT, il.quantity * p.amount)) AS Total
      FROM S23916715.Invoice_ProfG_FP AS i
               INNER JOIN S23916715.InvoiceLine_ProfG_FP AS il ON
i.id = il.invoice
               INNER JOIN S23916715.Price_ProfG_FP AS p ON
il.price = p.id
      WHERE YEAR(created) = YEAR(GETDATE())
        AND status != 'CANCELLED'
        AND deleted = 0) AS Invoices;
```

16. Get Total Paid this Year

This view gets all payment records for invoices that were created during the current year, then totals up the amount billed. The value is returned as a float. This view also includes payments on invoices that aren't 100% paid off (partial payments.)

```
CREATE OR ALTER VIEW S23916715.GetTotalPaidThisYear_ProfG_FP AS
SELECT CONVERT(FLOAT, SUM(CONVERT(BIGINT, Total))) / 100 AS
GrandTotal
FROM (SELECT SUM(CONVERT(BIGINT, amount)) AS Total
      FROM S23916715.InvoicePaymentRecord_ProfG_FP
      WHERE YEAR(date) = YEAR(GETDATE())) AS Records;
```

Deliverable 10 (Functions)

1. Get Invoice Status ID

Given the status name (as a `VARCHAR`), return the ID (PK) of the status

```
CREATE OR ALTER FUNCTION
S23916715.GetInvoiceStatusId_ProfF_FP(@status VARCHAR(128))
    RETURNS INT
AS
BEGIN
    -- Try to find the status by the value
    DECLARE @status_id INT;

SELECT @status_id = id FROM S23916715.InvoiceStatus_ProfG_FP WHERE
status = @status;

RETURN @status_id;

END
```

2. Get Payment Method ID

Given the payment method name (as a `VARCHAR`), return the ID (PK) of the `PaymentMethod`

```
CREATE OR ALTER FUNCTION
S23916715.GetPaymentMethodId_ProfF_FP(@payment_method
VARCHAR(128))
    RETURNS INT
AS
BEGIN
    -- Try to find the payment method by the value
    DECLARE @method_id INT;

SELECT @method_id = id FROM S23916715.PaymentMethod_ProfG_FP WHERE
```

```
method = @payment_method;

RETURN @method_id;

END
```

3. Calculate Remaining Balance

See Deliverable 4 #12 ("12. Calculate Remaining Balance" in "Deliverable 4 (Reports)")

4. Find Amount Paid

See Deliverable 4 #14 ("14. Find Amount Paid" in "Deliverable 4 (Reports)")

5. Validate Email

This function is a reusable function designed to simplify code in some logic functions. It takes an email address and checks if the email is valid and returns a `BIT`

```
CREATE OR ALTER FUNCTION S23916715.ValidateEmail_ProfG_FP(
    @email VARCHAR(128)
)
    RETURNS BIT
AS
BEGIN
    -- Valid by default

-- This function tries its best to check if a provided email is
valid
DECLARE @usernamePartLen INT;
DECLARE @domainPartLen INT;
DECLARE @pos INT;

-- Check if we have '@'
SET @pos = CHARINDEX('@', @email);
IF @pos = 0 OR @pos = LEN(@email)
    RETURN 0;
```

```
-- Make sure we don't have > 1 '@'
DECLARE @atCount INT;
SET @atCount = LEN(@email) - LEN(REPLACE(@email, '@', ''));

IF @atCount != 1
    RETURN 0;

-- Split by the '@' to extract the username and domain parts
SET @usernamePartLen = @pos - 1; -- One character to the left =
the length of the username (from beginning of str)
SET @domainPartLen = LEN(@email) - @pos;
-- The size of our entire string - the pos of '@' = the length of
our domain portion

-- Make sure we have minimum lengths
IF @usernamePartLen < 1 OR @domainPartLen < 3
    RETURN 0;

-- Check for invalid characters
IF CHARINDEX('`&\:;,\"-_<>[]()', @email) > 0
    RETURN 0;

-- Make sure the username portion doesn't contain any '@'s
IF CHARINDEX('@', SUBSTRING(@email, 0, @usernamePartLen)) > 0
    RETURN 0;

-- It's valid so return true
RETURN 1;

END
```

Deliverable 11 (Business Logic Procedures)

Initialize Invoice

This procedure takes the required information to create a new invoice.

Field	Description	Required
`customer_email`	The email of the customer associated with	✓
`invoice_num`	Override the automatically generated invoice number	❌
`due_date`	The date the invoice is due (mutually exclusive with `due_days`)	❌
`due_days`	How many days from the current date the invoice is due (mutually exclusive with `due_date`)	❌
`inserted_id`	Output var which returns the ID of the newly inserted invoice	❌

```
CREATE OR ALTER PROCEDURE S23916715.InitializeInvoice_ProfG_FP(
    @customer_email VARCHAR(320),
    @invoice_num INT = NULL,
    @description VARCHAR(1024) = NULL,
    @due_date DATE = NULL,
    @due_days INT = NULL, -- The amount of days (from today) which
the invoice is due
    @inserted_id INT OUT
)
```

```
AS
BEGIN
    SET NOCOUNT ON
    -- Step 1: Check that our customer exists
    DECLARE @customerID INT;

SELECT @customerID = id FROM S23916715.Customer_ProfG_FP WHERE
email = @customer_email;

IF @customerID IS NULL
    BEGIN
        PRINT CONCAT('Customer with email ', @customer_email, '
does not exist');
        RETURN
    END

-- Step 2a: If the user provided an invoice number, check if an
invoice already exists with this number
IF @invoice_num IS NOT NULL
    BEGIN
        DECLARE @foundInvoiceNum INT;

        SELECT @foundInvoiceNum = num FROM
S23916715.Invoice_ProfG_FP WHERE num = @invoice_num;

        IF @foundInvoiceNum IS NOT NULL
            BEGIN
                PRINT CONCAT('Invoice with the number ',
@foundInvoiceNum, ' exists')
                RETURN
            END
    END

-- Step 2b: If the user did not provide an invoice number, try to
increment it from our variables table
DECLARE @numVar VARCHAR(128);

SELECT @numVar = value FROM S23916715.Variables_ProfG_FP WHERE
```

```
[key] = 'invoice_accumulator';

IF @numVar IS NULL
    BEGIN
        -- We haven't started accumulating invoice numbers yet, so
start from 1
        INSERT INTO S23916715.Variables_ProfG_FP ([key], value)
VALUES ('invoice_accumulator', '1');
        SET @invoice_num = 1;
    END
ELSE
    BEGIN
        -- Convert the value to a INT
        SET @invoice_num = TRY_CAST(@numVar AS INT) + 1
        -- Now increment our accumulator in the database
        UPDATE S23916715.Variables_ProfG_FP
        SET value=TRY_CAST(@invoice_num AS VARCHAR)
        WHERE [key] = 'invoice_accumulator';
    END

-- Step 3: We can't have both the due_date and the due_days
IF @due_date IS NOT NULL AND @due_days IS NOT NULL
    BEGIN
        PRINT 'You can''t provide both due_date and due_days'
        RETURN
    END

-- Find the id for the 'DRAFT' status
DECLARE @statusId INT;

SELECT @statusId = id FROM S23916715.InvoiceStatus_ProfG_FP WHERE
status = 'DRAFT';
IF @statusId IS NULL
    BEGIN
        PRINT 'Something went wrong when creating new invoice (bad
status)'
        RETURN
    END
```

```
-- Step 4: Insert the new invoice
BEGIN TRY
    IF @due_days IS NOT NULL OR @due_date IS NOT NULL
        BEGIN
            DECLARE @due DATE;
            IF @due_date IS NOT NULL
                -- If we have the due_date, just insert that,
                SET @due = @due_date
            ELSE
                -- but if we have the due_days, set `due` =
today's date + the due_days
                SET @due = DATEADD(DAY, @due_days, GETDATE())

            INSERT INTO S23916715.Invoice_ProfG_FP (num, customer,
description, due, status)
            VALUES (@invoice_num, @customerID, @description, @due,
@statusId)
        END
    ELSE
        -- We we didn't provide either due_days or due_date, don't
insert it (it'll default to today + 30 days)
        BEGIN
            INSERT INTO S23916715.Invoice_ProfG_FP
                (num, customer, description, status)
            VALUES (@invoice_num, @customerID, @description,
@statusId);
        END

    -- Set our output variable to the ID of the last inserted ID
(scoped)
    SET @inserted_id = SCOPE_IDENTITY();
END TRY
BEGIN CATCH
    PRINT 'Something went wrong when initializing new invoice'
    RETURN
END CATCH
```

```
SET NOCOUNT OFF

END
```

Add Line to Invoice

This procedure adds a new product line to an invoice.

Field	Description	Required
`invoice_num`	The number of the invoice to add the line to	✓
`product_name`	The name of the product to add to the invoice (requires `product_sku`, mutually exclusive with `product_id`)	❌
`product_sku`	The sku of the product to add to the invoice (requires `product_name`, mutually exclusive with `product_id`)	❌
`product_id`	The id of the product to add to the invoice (mutually exclusive with `product_name` & `product_sku`)	❌
`price_id`	The id of the price to add to the line	✓
`quantity`	The quantity of the product to add to the invoice	✓
`description`	Any additional information associated with the line	❌
`inserted_id`	Output var which returns the id of the newly inserted invoice line	❌

```
CREATE OR ALTER PROCEDURE S23916715.AddNewLineToInvoice_ProfG_FP(
    @invoice_num INT,
    @product_name VARCHAR(128) = NULL,
    @product_sku VARCHAR(128) = NULL,
    @product_id INT = NULL,
    @price_id INT,
    @quantity INT,
    @description VARCHAR(1024) = NULL,
    @inserted_id INT OUT
)
AS
BEGIN
    SET NOCOUNT ON

-- Step 1: Make sure the invoice exists
DECLARE @invoice_id INT;

SELECT @invoice_id = id FROM S23916715.Invoice_ProfG_FP WHERE num
= @invoice_num;
IF @invoice_id IS NULL
    BEGIN
        PRINT CONCAT('Invoice with the number ', @invoice_num, '
doesn''t exist')
        RETURN
    END

-- Step 2: Check if we're going to find the product by id or
name&sku
-- We can't have both id and product&sku
IF @product_id IS NOT NULL AND @product_sku IS NOT NULL AND
@product_name IS NOT NULL
    BEGIN
        PRINT 'You can''t provide both product ID and product
name&sku'
        RETURN
    END
    -- We have to have at least one of the two
```

```
ELSE
    IF @product_id IS NULL AND (@product_name IS NULL OR
@product_sku IS NULL)
        BEGIN
            PRINT 'You must provide at least product ID or product
name&sku'
            RETURN
        END

-- Now we can try to find the product
SELECT @product_id = id
FROM S23916715.Product_ProfG_FP
WHERE (@product_id IS NOT NULL AND id = @product_id)
   OR (@product_name IS NOT NULL AND @product_sku IS NOT NULL AND
name = @product_name AND sku = @product_sku);

IF @@ROWCOUNT = 0
    BEGIN
        PRINT 'Product doesn''t exist'
        RETURN
    END

-- We can try to find the price
DECLARE @pricesProductId INT

SELECT @pricesProductId = product FROM S23916715.Price_ProfG_FP
WHERE id = @price_id;

IF @pricesProductId IS NULL
    BEGIN
        PRINT 'The provided price does not exist'
        RETURN
    END

-- The price must belong to the provided product
IF @pricesProductId != @product_id
    BEGIN
        PRINT 'The provided price does not belong to the provided
```

```
product'
        RETURN
    END

-- Validate our provided quantity
IF @quantity <= 0
    BEGIN
        PRINT 'Quantity must be > 0'
        RETURN
    END

-- Finally, insert our new invoice line
BEGIN TRY
    INSERT INTO S23916715.InvoiceLine_ProfG_FP (invoice, product,
price, quantity, description)
    VALUES (@invoice_id, @product_id, @price_id, @quantity,
@description);

    -- Set our output var to the new ID
    SET @inserted_id = SCOPE_IDENTITY();

    PRINT CONCAT('Successfully added line to in invoice num ',
TRY_CAST(@invoice_num AS VARCHAR))
END TRY
BEGIN CATCH
    PRINT 'Something went wrong when inserting new line into
invoice';
    RETURN
END CATCH

SET NOCOUNT OFF

END
```

Add Payment to Invoice

Log a payment for a given invoice. Automatically updates the invoice's status if necessary.

Field	Description	Required
`invoice_num`	The number of the invoice to log the payment to	✓
`amount`	The amount (in dollars and cents) paid	✓
`method`	The method of payment (check, cash, crypto, etc.)	✓
`check_num`	The number written on the check (only applicable if the `method` is "check")	❌
`payment_account`	The account the payment would be paid out to	✓

```
CREATE OR ALTER PROCEDURE S23916715.AddPaymentToInvoice_ProfG_FP(
    @invoice_num INT,
    @amount FLOAT,
    @method VARCHAR(20),
    @check_num VARCHAR(32) = NULL,
    @payment_account INT
)
AS
BEGIN
    SET NOCOUNT ON

-- Step 1: Make sure our invoice exists
DECLARE @invoice_id INT;

SELECT @invoice_id = id FROM S23916715.Invoice_ProfG_FP WHERE num
= @invoice_num;
```

```
IF @invoice_id IS NULL
    BEGIN
        PRINT CONCAT('Invoice with num ', @invoice_num, ' doesn''t
exist')
        RETURN
    END

-- Step 2: Validate the payment method the user provided
DECLARE @paymentMethodId INT;
SET @paymentMethodId =
S23916715.GetPaymentMethodId_ProfF_FP(@method);

IF @paymentMethodId IS NULL
    BEGIN
        PRINT CONCAT(@method, ' is an invalid payment method');
        RETURN
    END

-- Step 3: If the payment method isn't check but we provided a
check number, fail
IF @method != 'CHECK' AND @check_num IS NOT NULL
    BEGIN
        PRINT 'Check number can only be supplied when payment
method is check'
        RETURN
    END

-- Step 4: Make sure the payment account id is valid
DECLARE @findAccountCount INT;

SELECT @findAccountCount = COUNT(*)
FROM S23916715.PaymentAccount_ProfG_FP
WHERE id = @payment_account AND deleted = 0;
IF @findAccountCount IS NULL OR @findAccountCount = 0
    BEGIN
        PRINT 'Payment account doesn''t exist'
        RETURN
    END
```

```
-- Step 5: Start a transaction for creating the payment record
-- The reason we need to do this is because if the payment record
fully pays off the invoice,
-- we need to modify the invoice object as well, which could cause
problems if something goes wrong
-- mid-way trough
BEGIN TRANSACTION [Trans]
    BEGIN TRY
        -- Step 5a: Create the payment record
        INSERT INTO S23916715.InvoicePaymentRecord_ProfG_FP
(invoice, amount, method, check_num, payment_account)
        VALUES (@invoice_id, TRY_CAST(@amount * 100 AS INT),
@paymentMethodId,
                @check_num, @payment_account)

        DECLARE @remaining FLOAT;

        SET @remaining =
S23916715.CalculateRemainingBalance_ProfF_FP(@invoice_num)

        -- Step 5b: If the invoice is completely paid off, set the
status to paid and set the `paid` timestamp
        IF @remaining <= 0
            BEGIN
                UPDATE S23916715.Invoice_ProfG_FP
                SET status =
S23916715.GetInvoiceStatusId_ProfF_FP('PAID'),
                    paid=SYSDATETIME()
                WHERE num = @invoice_num;
            END

        -- Step 5c: Add the payment total to our account
        UPDATE S23916715.PaymentAccount_ProfG_FP
        SET balance = balance + TRY_CAST(@amount * 100 AS INT)
        WHERE id = @payment_account;

        COMMIT TRANSACTION [Trans]
```

```
    END TRY
    BEGIN CATCH
        PRINT 'Something went wrong when creating payment record'
        ROLLBACK TRANSACTION [Trans]
        RETURN
    END CATCH

    SET NOCOUNT OFF

END
```

Change Status on Several Invoices

Given several `invoice_nums` (in CSV format), set the `status` on each invoice to the given `@status`

Field	Description	Required
`status`	The status to set on each invoice	✓
`invoice_nums`	The invoice numbers to change the status of (in CSV format)	✓

```
CREATE OR ALTER PROCEDURE
S23916715.ChangeStatusOnSeveralInvoices_ProfG_FP(
    @status VARCHAR(128),
    @invoice_nums VARCHAR(1024) -- a csv of the invoice numbers
)
AS
BEGIN
    SET NOCOUNT ON

DECLARE @status_id INT;

-- Get (and validate) our status
```

```
SET @status_id = S23916715.GetInvoiceStatusId_ProfF_FP(@status);

IF @status_id IS NULL
    BEGIN
        PRINT CONCAT(@status, ' is an invalid status')
        RETURN
    END

-- Split the csv argument into a table for looping
CREATE TABLE #tableOfIds
(
    id INT
);

INSERT INTO #tableOfIds (id)
SELECT value
FROM STRING_SPLIT(@invoice_nums, ',');

-- Loop through each value in our temporary table and find the
invoice, then try to set its status
DECLARE @id INT;

BEGIN TRANSACTION [UpdateTransaction]

    BEGIN TRY
        WHILE EXISTS (SELECT * FROM #tableOfIds)
            BEGIN
                -- Pick an invoice num from the top of our temp
table
                SELECT TOP 1 @id = id FROM #tableOfIds;
                -- Delete that value so we don't pick it next
iteration
                DELETE FROM #tableOfIds WHERE id = @id;

                -- Make sure the invoice exists
                DECLARE @invoice_id INT;

                SELECT @invoice_id = id FROM
```

```
S23916715.Invoice_ProfG_FP WHERE num = @id;

                IF @invoice_id IS NULL
                    BEGIN
                        PRINT CONCAT('Invoice with num ', @id, '
doesn''t exist')
                        RETURN
                    END

                -- Now that we know it exists, update its status
                UPDATE S23916715.Invoice_ProfG_FP SET
status=@status_id WHERE num = @id;

            END

        -- Remove our temporary table
        DROP TABLE #tableOfIds;
    END TRY
    BEGIN CATCH
        PRINT 'Something went wrong when updating invoice'
        ROLLBACK TRANSACTION [UpdateTransaction]
        RETURN
    END CATCH
COMMIT TRANSACTION [UpdateTransaction]

SET NOCOUNT OFF

END
```

Create New Customer

Field	Description	Required
email	The customer's email address	✓
phone_num	The customer's phone number	✓
first_name	The customer's first name	✓
last_name	The customer's last name	✓
address	Address line one	✓
address_line_two	Address line two	❌
city	City	✓
state	State (2 characters)	✓
zip_code	Zip code	✓
inserted_id	Output parameter for inserted ID	❌

```
CREATE OR ALTER PROCEDURE S23916715.CreateNewCustomer_ProfG_FP(
    @email VARCHAR(320),
    @phone_num VARCHAR(12),
    @first_name VARCHAR(128),
    @last_name VARCHAR(128),
    @address VARCHAR(128),
    @address_line_two VARCHAR(128) = NULL,
    @city VARCHAR(128),
    @state CHAR(2),
    @zip_code VARCHAR(16),
    @inserted_id INT OUT
)
```

```
AS
BEGIN
    SET NOCOUNT ON

-- Step 1: Make sure we don't already have this customer (the
email should be unique)
DECLARE @findCustomerCount INT;

SELECT @findCustomerCount = COUNT(*) FROM
S23916715.Customer_ProfG_FP WHERE email = @email;
IF @findCustomerCount IS NOT NULL AND @findCustomerCount != 0
    BEGIN
        PRINT CONCAT('Customer with email ', @email, ' already
exists!');
        RETURN
    END

-- Step 2: Check if the given email is valid
DECLARE @isValid BIT;
SET @isValid = S23916715.ValidateEmail_ProfG_FP(@email);

IF @isValid = 0
    BEGIN
        PRINT 'Please provide a valid email address'
        RETURN
    END

-- Step 3: Make sure all of the provided fields aren't null
-- Note: We don't need to check the email since our validate
function does it
IF @phone_num IS NULL OR
   @first_name IS NULL OR
   @last_name IS NULL OR
   @address IS NULL OR
   @city IS NULL OR
   @state IS NULL OR
   @zip_code IS NULL
    BEGIN
```

```
        PRINT 'Please provide all required fields'
        RETURN
    END


-- At this point we can insert our customer into our database
BEGIN TRY
    INSERT INTO S23916715.Customer_ProfG_FP
    (email,
     phone_num,
     first_name,
     last_name,
     address,
     address_line_two,
     city, state, zip_code)
    VALUES (@email,
            @phone_num,
            @first_name,
            @last_name,
            @address,
            @address_line_two,
            @city, @state, @zip_code)

    -- Set our output variable to the ID of the last inserted ID
(scoped)
    SET @inserted_id = SCOPE_IDENTITY();

    PRINT 'Successfully added new customer'
END TRY
BEGIN CATCH
    PRINT 'Something went wrong when creating new customer';
    RETURN
END CATCH


SET NOCOUNT OFF
```

```
END
```

Create New Product

Field	Description	Required
name		✓
description	The default description of the product	❌
sku	The sku of the product	✓
inserted_id	Output parameter for inserted ID	✓

```
CREATE OR ALTER PROCEDURE S23916715.CreateNewProduct_ProfG_FP(
    @name VARCHAR(128),
    @description VARCHAR(1204) = NULL,
    @sku VARCHAR(128),
    @inserted_id INT OUT
)
AS
BEGIN
    SET NOCOUNT ON

-- Step 1: Make sure we don't already have a product with this
name or sku
DECLARE @findProductCount INT;

-- The name and the sku field don't need to be unique on their
own, but the combination needs to be
SELECT @findProductCount = COUNT(*) FROM
S23916715.Product_ProfG_FP WHERE name = @name AND sku = @sku;
IF @findProductCount IS NOT NULL AND @findProductCount != 0
    BEGIN
```

```
        PRINT CONCAT('Product with the name ', @name, ' and sku ',
@sku, ' already exist');
        RETURN
    END

-- Step 2: Make sure we have all of the required fields
IF @name IS NULL OR
   @sku IS NULL
    BEGIN
        PRINT 'Please provide all required fields'
        RETURN
    END

BEGIN TRY
    INSERT INTO S23916715.Product_ProfG_FP
        (name, description, sku)
    VALUES (@name,
            @description,
            @sku)

    -- Set our output variable to the ID of the last inserted ID
(scoped)
    SET @inserted_id = SCOPE_IDENTITY();

    PRINT 'Successfully added new product'

END TRY
BEGIN CATCH
    PRINT 'Something went wrong when creating new product';
    RETURN
END CATCH

SET NOCOUNT OFF

END
```

Deliverable 12 (Triggers)

Override Customer Delete

According to the companies data retention policy, no data can be deleted. To solve this, the `DELETE` operation is overwritten and replace with setting the `deleted` column to true.

```
CREATE OR ALTER TRIGGER S23916715.DeleteCustomer_Trigger_ProfG_FP
    ON S23916715.Customer_ProfG_FP
    INSTEAD OF DELETE
    AS
BEGIN
    SET NOCOUNT ON;

UPDATE S23916715.Customer_ProfG_FP
SET deleted = 1
FROM S23916715.Customer_ProfG_FP
         INNER JOIN deleted ON S23916715.Customer_ProfG_FP.id =
deleted.id;

SET NOCOUNT OFF;

END;
```

Override Invoice Delete

According to the companies data retention policy, no data can be deleted. To solve this, the `DELETE` operation is overwritten and replace with setting the `deleted` column to true.

```
CREATE OR ALTER TRIGGER
S23916715.OverrideDeleteInvoice_Trigger_ProfG_FP
    ON S23916715.Invoice_ProfG_FP
    INSTEAD OF DELETE
    AS
```

```
BEGIN
    SET NOCOUNT ON;

UPDATE S23916715.Invoice_ProfG_FP
SET deleted = 1, updated=SYSDATETIME()
FROM S23916715.Invoice_ProfG_FP
         INNER JOIN deleted ON S23916715.Invoice_ProfG_FP.id =
deleted.id;

SET NOCOUNT OFF;

END;
```

INSERT/UPDATE/DELETE Trigger for Invoice

This trigger runs on all `INSERT`, `UPDATE`, and `DELETE` operations, but has different behavior for each event. On insert, the associated customer's invoice count is incremented. On update, the invoice's update field is set to the current timestamp On delete, the invoice's deleted field is set to true and the update field is set to the current timestamp

```
CREATE OR ALTER TRIGGER S23916715.MultiTriggerForInvoice_ProfG_FP
    ON S23916715.Invoice_ProfG_FP
    AFTER INSERT, UPDATE, DELETE
    AS
BEGIN
    SET NOCOUNT ON;

DECLARE @eventType VARCHAR(10);

-- Grab the event type so we can run some code conditionally
SET @eventType = CASE
                     WHEN EXISTS (SELECT * FROM INSERTED) THEN
'INSERT'
                     WHEN EXISTS (SELECT * FROM DELETED) THEN
'DELETE'
                     ELSE 'UPDATE'
```

```
    END;

IF @eventType = 'INSERT'
    BEGIN
        -- When we insert new records, find the customer and
increment their invoice count
        DECLARE @insertedRows INT = @@ROWCOUNT;

        IF @insertedRows > 0
            BEGIN
                UPDATE S23916715.Customer_ProfG_FP
                SET invoice_count = invoice_count + 1
                FROM S23916715.Invoice_ProfG_FP AS inv
                        INNER JOIN S23916715.Customer_ProfG_FP AS
cust ON inv.customer = cust.id;
            END
    END

ELSE
    IF @eventType = 'UPDATE'
        -- When we update an invoice, set its `updated` field
        BEGIN
            UPDATE S23916715.Invoice_ProfG_FP
            SET updated = SYSDATETIME()
            FROM S23916715.Invoice_ProfG_FP
                    INNER JOIN inserted ON
S23916715.Invoice_ProfG_FP.id = inserted.id;
        END

    ELSE
        IF @eventType = 'DELETE'
            -- When we delete an invoice, instead mark it as
deleted (and set its updated field)
            BEGIN
                UPDATE S23916715.Invoice_ProfG_FP
                SET deleted = 1 -- Set a flag indicating that the
file was marked 'deleted'
                FROM deleted AS d
```

```
                WHERE S23916715.Invoice_ProfG_FP.id = d.id;
            END

SET NOCOUNT OFF;

END
```

Customer Update

On update, the customer's updated field is set to the current timestamp

```
CREATE OR ALTER TRIGGER S23916715.CustomerUpdateTrigger_ProfG_FP
    ON S23916715.Customer_ProfG_FP
    AFTER UPDATE
    AS
BEGIN
    SET NOCOUNT ON;

-- When we update a customer, set its `updated` field
UPDATE S23916715.Customer_ProfG_FP
SET updated = SYSDATETIME()
FROM S23916715.Customer_ProfG_FP
         INNER JOIN inserted ON S23916715.Customer_ProfG_FP.id =
inserted.id;

SET NOCOUNT OFF;

END
```

Insert Trigger for Invoice Line

When adding a new line to an invoice, increment that invoice's line count

```
CREATE OR ALTER TRIGGER
S23916715.InsertTriggerForInvoiceLine_ProfG_FP
    ON S23916715.InvoiceLine_ProfG_FP
    AFTER INSERT
```

```
    AS
BEGIN
    SET NOCOUNT ON;

-- When we insert a new invoice line, update the invoice's line
count

DECLARE @insertedRows INT = @@ROWCOUNT;

IF @insertedRows > 0
    BEGIN
        UPDATE S23916715.Invoice_ProfG_FP
        SET lines = lines + 1
        FROM S23916715.Invoice_ProfG_FP
                 INNER JOIN inserted ON
S23916715.Invoice_ProfG_FP.id = inserted.invoice
        WHERE S23916715.Invoice_ProfG_FP.id = inserted.id;
    END

SET NOCOUNT OFF;

END;
```

Delete Payment Account

When the payment account is deleted, instead mark it as deleted

```
CREATE OR ALTER TRIGGER
S23916715.OverrideDeletePaymentAccount_Trigger_ProfG_FP
    ON S23916715.PaymentAccount_ProfG_FP
    INSTEAD OF DELETE
    AS
BEGIN
    SET NOCOUNT ON;

UPDATE S23916715.PaymentAccount_ProfG_FP
SET deleted = 1
```

```
FROM S23916715.PaymentAccount_ProfG_FP
         INNER JOIN deleted ON
S23916715.PaymentAccount_ProfG_FP.id = deleted.id;

SET NOCOUNT OFF;

END;
```

Insert Invoice Line

When inserting an invoice line, update its invoice's line count

```
CREATE OR ALTER TRIGGER
S23916715.InsertTriggerForInvoiceLine_ProfG_FP
    ON S23916715.InvoiceLine_ProfG_FP
    AFTER INSERT
    AS
BEGIN
    SET NOCOUNT ON;

-- When we insert a new invoice line, update the invoice's line
count

DECLARE @insertedRows INT = @@ROWCOUNT;

IF @insertedRows > 0
    BEGIN
        UPDATE S23916715.Invoice_ProfG_FP
        SET lines = lines + 1
        FROM S23916715.Invoice_ProfG_FP
                 INNER JOIN inserted ON
S23916715.Invoice_ProfG_FP.id = inserted.invoice
        WHERE S23916715.Invoice_ProfG_FP.id = inserted.id;
    END

SET NOCOUNT OFF;
```

```
END;
```

Update Invoice Payment Record

When updating an invoice payment record, we need to make sure that the balance on the associated payment account matches. To solve this, after updating a payment record, we subtract the original value from the account and then add the new value

```
CREATE OR ALTER TRIGGER
S23916715.UpdateTriggerForInvoicePaymentRecord_ProfG_FP
    ON S23916715.InvoicePaymentRecord_ProfG_FP
    AFTER UPDATE
    AS
BEGIN
    SET NOCOUNT ON;

-- We don't need to use a transactions because triggers are
already run in the context of a transaction
-- When we update a invoice payment record, we need to correct our
total balance in our payment account
BEGIN TRY
    -- Subtract the old amount
    UPDATE S23916715.PaymentAccount_ProfG_FP
    SET balance = balance - (SELECT amount FROM deleted)
    WHERE id = (SELECT payment_account FROM deleted);

    -- Add the new amount
    UPDATE S23916715.PaymentAccount_ProfG_FP
    SET balance = balance + (SELECT amount FROM inserted)
    WHERE id = (SELECT payment_account FROM inserted);

END TRY
BEGIN CATCH
    PRINT 'Something went wrong when updating payment account'
    RETURN
END CATCH
```

```
SET NOCOUNT OFF;

END
```