Table of Contents

Capstone Walkthrough Part 1

Cleaned **cdw_sapp_branch** to **cdw_sapp_branch_cleaned**.

To ensure data quality and consistency, I first transformed the branch data using Power Query.

- On **BRANCH_ZIP** I first changed the data type to text. I used **Custom Column** with the formula

  ```
  if Text.Length([BRANCH_ZIP]) = 4 then "0" & [BRANCH_ZIP] else [BRANCH_ZIP]
  ```

  so if the zip only returned four characters it would add a 0. If the zip returned five characters it will print the full zip.
- For **BRANCH_CITY** I used **Column From Example** to add a space between each word until the rest auto-filled.
- On **BRANCH_PHONE** I used **Column From Example** to change the format to (123) 456-5276 until the rest auto-filled.
- On **LAST_UPDATED** I changed the format to **Date/Time**.

Once the three columns were created I replaced them with the old columns.

Cleaned **cdw_sapp_customer** to **cdw_sapp_customer_cleaned**.

The customer data required several cleaning steps to standardize names, format personal information correctly, and prepare it for analysis.

- Titles such as Mr., Mrs., and Ms. were removed from **FIRST_NAME** using a **Custom Column** with the following formula and the original column was replaced.

  ```
  Text.Trim(Text.Replace(Text.Replace(Text.Replace(Text.Replace(Text.Replace(Text.Replace([FIRST_NAME], "Mr ", ""), "Mr. ", ""), "Mrs ", ""), "Mrs. ", ""), "Ms ", ""), "Ms. ", ""))
  ```

- Some **FIRST_NAME** values contained a last name, so I used Split Column by Delimiter, using a space, and deleted the extra columns (**FIRST_NAME.2**, **FIRST_NAME.3**, **FIRST_NAME.4**) that were created.
- For **SSN**, the data type was set to **Text**, then formatted using **Column From Example** by manually adding hyphens between the third and sixth digits until Power Query auto-filled the remaining rows. The formatted column replaced the original.
- **CREDIT_CARD_NO** was converted to **Text** and reformatted with **Column From Example** by adding a space every four digits (for example, 1234 5678 9012 3456). The new column replaced the old column.
- **CUST_CITY** was standardized using **Column From Examples**, where I typed each name with proper spacing until all rows auto-filled, then replaced the old column.
- On **CUST_ZIP**, I changed the data type to text and applied a **Custom Column** formula

```
if Text.Length([CUST_ZIP]) = 4 then "0" & [CUST_ZIP] else [CUST_ZIP]
```

  so that any four-digit zip codes were corrected. The updated column replaced the original.
- For **CUST_PHONE**, I used **Column From Example** to format the number as 123-7017. After that, a **Custom Column** was added with the formula

```
"(555) " & [CUST_PHONE]
```

  to append the area code, and the formatted column replaced the original.

Cleaned **cdw_sapp_loan_application** **to** **cdw_sapp_loan_application_cleaned.**

For the loan application dataset, only one change was needed.

- For **Income** I changed the format to **Capitalize Each Word** to keep values consistent.

Cleaned cdw_sapp_transaction to cdw_sapp_transaction_cleaned.

For the transaction data, the focus was on ensuring data types were correct for accurate financial calculations and analysis.

- **CREDIT_CARD_NO** was converted to **Text** and reformatted with **Column From Example** by adding a space every four digits (for example, 1234 5678 9012 3456). The new column replaced the old column.
- For **CUST_SSN**, the data type was set to **Text**, then formatted using **Column From Example** by manually adding hyphens between the third and sixth digits until Power Query auto-filled the remaining rows. The formatted column replaced the original.
- For **Transaction_Amount** I changed the format to **Currency**.
- For **Time**, I changed the data type to **Date/Time** so it matches the format on **Branch** data.

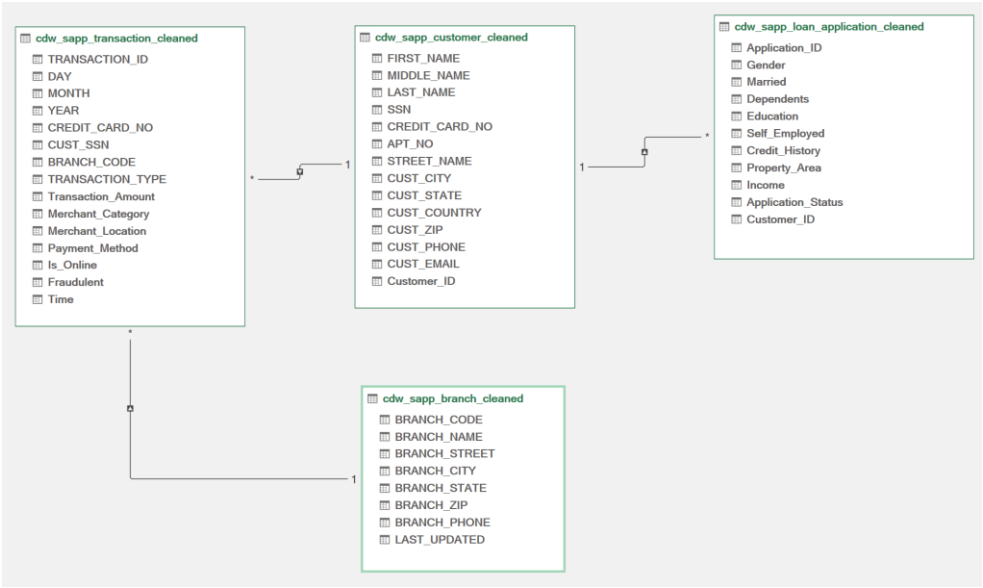
Functional Requirement 1.1

I imported all the cleaned datasets into Excel using **Only Create Connection** and adding it to the Data Model.

Created **one-to-many** relationships between datasets

- **cdw_sapp_customer[SSN]** -> **cdw_sapp_transaction[CUST_SSN]**
- **cdw_sapp_loan_application[Customer_ID]** -> **cdw_sapp_customer[Customer_ID]**
- **cdw_sapp_branch[BRANCH_CODE]** -> **cdw_sapp_transaction[BRANCH_CODE]**

This structured star schema is essential for accurate analysis, as it ensures that all tables are correctly linked and that data integrity is maintained across the entire model.



Functional Requirement 1.2

I first wanted to drop **creditcard_capstone** so if the whole code is run it will create a brand new database.

```
DROP DATABASE IF EXISTS creditcard_capstone; -- deletes creditcard_capstone
```

Created the database **creditcard_capstone** in MySQL.

```
CREATE DATABASE creditcard_capstone; -- create the database creditcard_capstone
```

Made the database the default schema.

```
USE creditcard_capstone; -- make creditcard_capstone the default schema for the rest of this script
```

Created two tables in the MySQL script, **cdw_sapp_customers** and **cdw_sapp_transaction**.

```
CREATE TABLE cdw_sapp_customers ( -- create a new table called
cdw_sapp_customers
   FIRST_NAME VARCHAR(50) NOT NULL, -- customer's first name which is required
   MIDDLE_NAME VARCHAR(50), -- customer's middle name, optional since not
everyone has one
   LAST_NAME VARCHAR(50) NOT NULL, -- customer's last name which is required
   SSN VARCHAR(11) PRIMARY KEY NOT NULL, -- social security number used as a
unique identifier (primary key) which is required
   CREDIT_CARD_NO VARCHAR(16) UNIQUE NOT NULL, -- credit card number for each
customer, unique to prevent duplicates which is required
   APT_NO VARCHAR(10), -- apartment or unit number, optional
   STREET_NAME VARCHAR(100) NOT NULL, -- street name for the customer's address
which is required
   CUST_CITY VARCHAR(50) NOT NULL, -- city where the customer lives which is
required
   CUST_STATE VARCHAR(50) NOT NULL, -- state where the customer lives which is
required
   CUST_COUNTRY VARCHAR(50) NOT NULL, -- country for customer location which is
required
   CUST_ZIP VARCHAR(10) NOT NULL, -- zip code as text so leading zeros don't get
removed which is required
   CUST_PHONE VARCHAR(15), -- customer phone number stored as text to keep
consistent formatting
   CUST_EMAIL VARCHAR(100), -- customer email address
   Customer_ID VARCHAR(100) -- customer id field used for reference (not primary
key)
);
```

```
CREATE TABLE cdw_sapp_transaction ( -- create a new table
called cdw_sapp_transaction
 TRANSACTION_ID INT NOT NULL, -- unique transaction number for
each record which is required
 `DAY` INT NOT NULL, -- day value for when the transaction
happened which is required
 `MONTH` INT NOT NULL, -- month value for the transaction date
which is required
 `YEAR` INT NOT NULL, -- year value for the transaction date
which is required
 CREDIT_CARD_NO VARCHAR(16) NOT NULL, -- credit card number
used in the transaction which is required
 CUST_SSN VARCHAR(11) NOT NULL, -- ssn that connects this
transaction back to a customer in cdw_sapp_customers which is
required
 BRANCH_CODE VARCHAR(10) NOT NULL, -- code for which branch
processed this transaction which is required
 TRANSACTION_TYPE VARCHAR(50) NOT NULL, -- type of transaction
(shopping, healthcare, education, etc.) which is required
 TRANSACTION_VALUE_OLD DECIMAL(10, 2) NOT NULL, -- old column
from original data that does not exist in cleaned version (will
be dropped later) which is required
 TRANSACTION_AMOUNT DECIMAL(10, 2) NOT NULL, -- actual
transaction amount in dollars which is required
 MERCHANT_CATEGORY VARCHAR(50) NOT NULL, -- category of the
merchant for this transaction which is required
MERCHANT_LOCATION VARCHAR(100) NOT NULL, -- city or location
where the merchant is based which is required
 PAYMENT_METHOD VARCHAR(50) NOT NULL, -- payment method
(credit, online, etc.) which is required
  IS_ONLINE BOOLEAN NOT NULL, -- whether the transaction was
done online (true or false) which is required
 FRAUDULENT BOOLEAN NOT NULL, -- whether this transaction was
flagged as fraudulent which is required
 `Time` TIME -- time of the transaction
);
```

Once the two tables were created I loaded the data into MySQL.

```
LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/cdw_sapp_customer_cleaned.csv' -- path to the csv file being imported
INTO TABLE cdw_sapp_customers -- specify which table to load data into
FIELDS TERMINATED BY ',' -- each column value in the csv is separated by a comma
OPTIONALLY ENCLOSED BY '"' -- keep text values inside quotes as single entries
LINES TERMINATED BY '\r\n' -- rows in csv end with windows-style newline
IGNORE 1 ROWS -- skip header row
(FIRST_NAME, MIDDLE_NAME, LAST_NAME, SSN, CREDIT_CARD_NO, APT_NO, STREET_NAME,
 CUST_CITY, CUST_STATE, CUST_COUNTRY, CUST_ZIP, CUST_PHONE, CUST_EMAIL, Customer_ID); -- list of columns in order matching the csv file
```

```
LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/cdw_sapp_transaction_cleaned.csv' -- path to the cleaned transaction csv file
INTO TABLE cdw_sapp_transaction -- specify which table to load data into
FIELDS TERMINATED BY ',' -- each column value in the csv is separated by a comma
OPTIONALLY ENCLOSED BY '"' -- keep text values inside quotes as single entries
LINES TERMINATED BY '\r\n' -- rows in csv end with windows-style newline
IGNORE 1 ROWS -- skip header row
(TRANSACTION_ID, `DAY`, `MONTH`, `YEAR`, CREDIT_CARD_NO, CUST_SSN, BRANCH_CODE,
 TRANSACTION_TYPE, TRANSACTION_VALUE_OLD, TRANSACTION_AMOUNT, MERCHANT_CATEGORY,
 MERCHANT_LOCATION, PAYMENT_METHOD, IS_ONLINE, FRAUDULENT, `TIME`); -- list of columns in order matching the csv file
```

At this point, running the script all at once can halt on the initial attempt because **TRANSACTION_VALUE_OLD** isn't in the CSV. To let the walkthrough demonstrate the initial load while still allowing a full run, I temporarily relaxed the session's strict mode (warnings instead of errors).

```
SET @old_sql_mode := @@SESSION.sql_mode; -- remember current session sql_mode
SET SESSION sql_mode := REPLACE(@@SESSION.sql_mode, 'STRICT_TRANS_TABLES', ''); -- allow warnings instead of errors for this load
SET SESSION sql_mode := REPLACE(@@SESSION.sql_mode, 'STRICT_ALL_TABLES', ''); -- ensure non-strict behavior for the initial load
```

I then ran the code to load in the **Transaction** table. Once that code was run I return the original strictness so later queries behave normally.

```
SET SESSION sql_mode := @old_sql_mode; -- restore the original session mode
```

Now the script will run no matter if you run all lines or run lines step-by-step.

Next, **TRANSACTION_VALUE_OLD** will be dropped from the table.

```
ALTER TABLE cdw_sapp_transaction -- drop extra column so csv columns align with table
DROP COLUMN TRANSACTION_VALUE_OLD; -- prevents values like 'clothing' from shifting into transaction_amount
```

After that the column **TIME** will change format to VARCHAR(20) to be able to load the time from the csv file as is.

```
ALTER TABLE cdw_sapp_transaction -- change time to text so csv can load full date and time
MODIFY COLUMN `Time` VARCHAR(20); -- allows values like 10/14/2025 6:24 to load as-is
```

I added a line of code so if a load command is run multiple times it will not append the data to make duplicates of everything.

```
TRUNCATE TABLE cdw_sapp_transaction; -- clear rows from the initial attempt so the corrected reload does not append
```

Now the data for **cdw_sapp_transaction** will again be loaded into MySQL, this time without the column **TRANSACTION_VALUE_OLD**.

```
LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/cdw_sapp_transaction_cleaned.csv' -- path to transaction csv after schema fixes
INTO TABLE cdw_sapp_transaction -- table now matches csv exactly
FIELDS TERMINATED BY ','  -- csv is comma separated
OPTIONALLY ENCLOSED BY '"'  -- allow quoted fields
LINES TERMINATED BY '\r\n'  -- windows newline
IGNORE 1 ROWS  -- skip header
(TRANSACTION_ID, `DAY`, `MONTH`, `YEAR`, CREDIT_CARD_NO, CUST_SSN, BRANCH_CODE,
 TRANSACTION_TYPE, TRANSACTION_AMOUNT, MERCHANT_CATEGORY, MERCHANT_LOCATION,
 PAYMENT_METHOD, IS_ONLINE, FRAUDULENT, `TIME`); -- column list matches csv order
```

Next, **TIME** will be converted from text to datetime.

```
UPDATE cdw_sapp_transaction -- convert text date and time into mysql datetime
SET `Time` = STR_TO_DATE(`Time`, '%m/%d/%Y %k:%i'); -- becomes 'YYYY-MM-DD HH:MM:SS'
```

Lastly **TIME** will be formatted as DATETIME to show the date exactly how it is in the original dataset. This will make sure all time functions work.

```
ALTER TABLE cdw_sapp_transaction -- convert time column to DATETIME for accurate time-based analysis
MODIFY COLUMN `Time` DATETIME; -- enables use of functions like HOUR(), MINUTE(), and time-based grouping in future queries
```

With that completed running **SELECT** statements will verify that all data has been pulled into MySQL.

```
SELECT * FROM cdw_sapp_customers; -- quick validation of customer load
```

FIRST_NAME	MIDDLE_NAME	LAST_NAME	SSN	CREDIT_CARD_NO	APT_NO	STREET_NAME	CUST_CITY	CUST_STATE	CUST_COUNTRY	CUST_ZIP	CUST_PHONE	CUST_EMAIL	Customer_ID
Molly	Andreas	Camp	123-45-1007	4210653344660820	762	College Avenue	Chambersburg	PA	United States	17201	(555) 123-1782	MCamp@example.com	C0574
Steven	Garfield	Fields	123-45-1012	4210653375981320	419	Sycamore Drive	San Lorenzo	CA	United States	94580	(555) 123-4687	SFields@example.com	C0239
Mark	Bernadette	Finn	123-45-1037	4210653316827500	389	Winding Way	Carrollton	GA	United States	30117	(555) 123-3309	OFinn@example.com	C0867
Johnathan	Logan	Lane	123-45-1041	4210653352152800	489	Henry Street	Reidsville	NC	United States	27320	(555) 123-7396	ILane@example.com	C0493
Jill	Austin	Richards	123-45-1068	4210653312588690	147	Durham Road	Rowlett	TX	United States	75088	(555) 123-6077	TRichards@example.com	C0922
James	Abel	Campbell	123-45-1069	4210653376538490	561	Chestnut Street	Victoria	TX	United States	77904	(555) 123-0528	LCampbell@example.com	C0237
Brandon	Tania	Deleon	123-45-1070	4210653327270060	500	Holly Drive	Osseo	MN	United States	55311	(555) 123-0421	RDeleon@example.com	C0759
Elizabeth	Erin	Parks	123-45-1071	4210653314009560	84	Broad Street	Trenton	NJ	United States	08610	(555) 123-5784	DParks@example.com	C0902
Christy	Agnes	Womack	123-45-1088	4210653379290310	656	East Avenue	New Haven	CT	United States	06511	(555) 123-2026	AWomack@example.com	C0214
Lindsey	Joyce	Lugo	123-45-1092	4210653322807170	724	Route 41	Jonesboro	GA	United States	30236	(555) 123-9010	NLugo@example.com	C0807
Christine	Cyrus	Hines	123-45-1093	4210653389763990	30	Mulberry Lane	Plainview	NY	United States	11803	(555) 123-8634	KHines@example.com	C0088
Stephen	Clifford	Mcconnell	123-45-1104	4210653371997790	116	Aspen Court	New Haven	CT	United States	06511	(555) 123-1487	JMcconnell@example.com	C0271
Ashley	Pearlie	Chase	123-45-1105	4210653337579460	401	Spruce Street	New Berlin	WI	United States	53151	(555) 123-3104	FChase@example.com	C0650
Sean	Huey	Doss	123-45-1108	4210653383788450	158	South Street	Acworth	GA	United States	30101	(555) 123-3066	HDoss@example.com	C0157
Ryan	Leticia	Daly	123-45-1125	4210653379686110	717	Brandywine Drive	Grandville	MI	United States	49418	(555) 123-1370	TDaly@example.com	C0202
Kevin	Arnoldo	Billings	123-45-1131	4210653365921960	716	Schoolhouse Lane	Bettendorf	IA	United States	52722	(555) 123-9862	LBillings@example.com	C0330

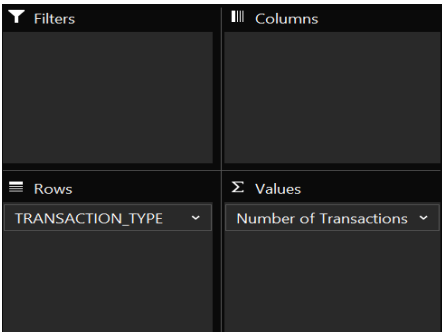
```
SELECT * FROM cdw_sapp_transaction; -- quick validation of transaction load
```

TRANSACTION_ID	DAY	MONTH	YEAR	CREDIT_CARD_NO	CUST_SSN	BRANCH_CODE	TRANSACTION_TYPE	TRANSACTION_VALUE_OLD	TRANSACTION_AMOUNT	MERCHANT_CATEGORY	MERCHANT_LOCATION	PAYMENT_METHOD
1	14	2	2018	4210653349028690	123-45-9988	114	Education	NULL	1000.20	Clothing	New Aliciamouth	Debit Card
2	20	3	2018	4210653349028690	123-45-9988	35	Entertainment	NULL	3234.10	Health	Butlerville	Credit Card
3	8	7	2018	4210653349028690	123-45-9988	160	Grocery	NULL	1258.35	Health	Port Wendybury	Debit Card
4	19	4	2018	4210653349028690	123-45-9988	114	Entertainment	NULL	4615.70	Dining	South Debra	Credit Card
5	10	10	2018	4210653349028690	123-45-9988	93	Gas	NULL	4918.14	Groceries	Ericafort	Debit Card
6	28	5	2018	4210653349028690	123-45-9988	164	Education	NULL	1009.00	Electronics	Lake Maryside	Credit Card
7	19	5	2018	4210653349028690	123-45-9988	119	Entertainment	NULL	276.57	Electronics	Brownberg	Credit Card
8	8	8	2018	4210653349028690	123-45-9988	23	Gas	NULL	205.08	Electronics	Yvettechester	Debit Card
9	18	3	2018	4210653349028690	123-45-9988	166	Entertainment	NULL	1995.63	Electronics	Powellbury	Credit Card
10	3	9	2018	4210653349028690	123-45-9988	83	Bills	NULL	4072.53	Travel	Charleneside	Debit Card
11	21	8	2018	4210653349028690	123-45-9988	52	Gas	NULL	1856.48	Travel	Nguyenton	Credit Card
12	24	12	2018	4210653349028690	123-45-9988	17	Gas	NULL	4146.79	Clothing	Kellystad	Debit Card
13	3	4	2018	4210653349028690	123-45-9988	80	Grocery	NULL	761.97	Clothing	Stacymouth	Credit Card
14	15	4	2018	4210653349028690	123-45-9988	50	Bills	NULL	1034.72	Health	North Victoriaberg	Debit Card
15	17	5	2018	4210653349028690	123-45-9988	123	Bills	NULL	4365.23	Groceries	West Andrewmouth	Credit Card

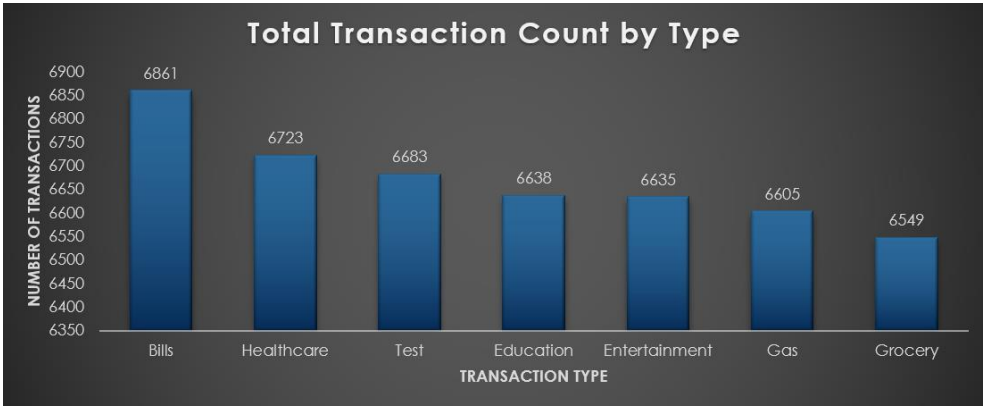
Functional Requirement 3.1

To calculate the total count for each transaction type I created a PivotTable from the transaction dataset to count the number of transactions for each transaction type. I then sorted largest to smallest.

Transaction Type	Number of Transactions
Bills	6861
Healthcare	6723
Test	6683
Education	6638
Entertainment	6635
Gas	6605
Grocery	6549
Grand Total Transactions	**46694**



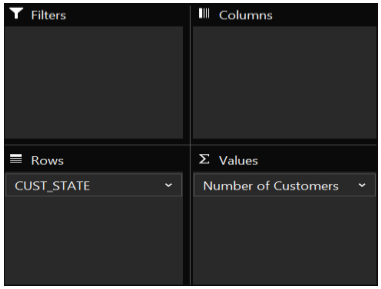
I then took that data and created a **Clustered Column Chart** to show which transaction type occurred the most often. **Bills** was the transaction type with the most transactions at **6861** while **Grocery** was the lowest with **6549**. This indicates that recurring, essential expenses are the primary driver of transaction volume in the dataset.



Functional Requirement 3.2

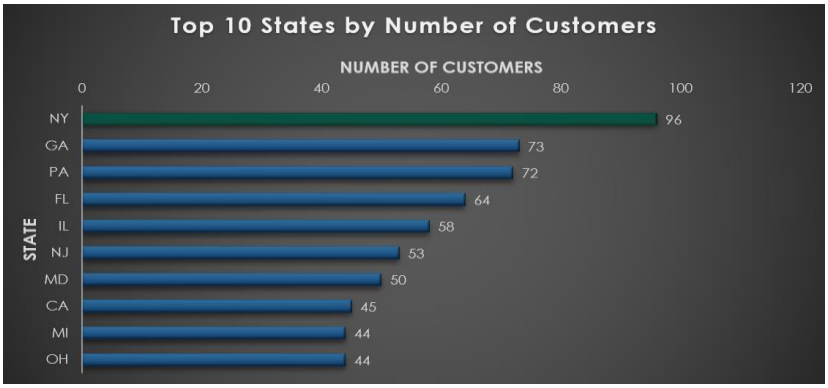
To calculate the top 10 states with the highest number of customers I created a PivotTable from the customer dataset and looked at States with a count of **Customer_ID**. I then told it to only show the top 10 based on that Customer count.

State	Number of Customers
NY	96
GA	73
PA	72
FL	64
IL	58
NJ	53
MD	50
CA	45
MI	44
OH	44
Grand Total	**599**



To best show this information in a visual I chose a **Clustered Bar Chart**. Of the top 10 it shows **NY** as having the most customers with **96** and **OH** has the least amount of customers with **44**.

This analysis identifies the company's key markets, highlighting New York as the largest customer base and providing a clear focus for regional marketing strategies.

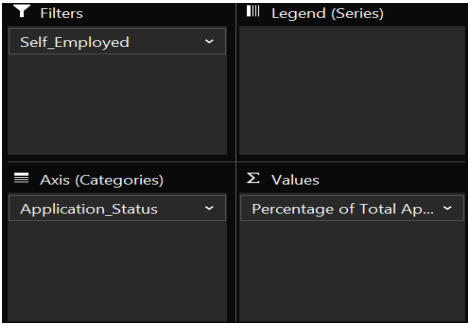


Functional Requirement 3.3

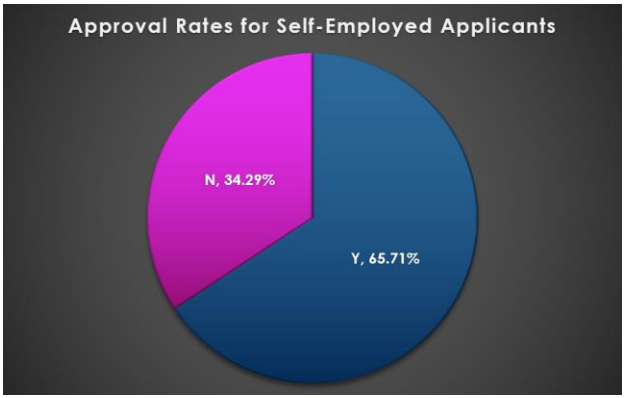
To calculate the percentage of applications that were approved for self-employed applicants I created a PivotTable from the loan application dataset. I created a filter for **Self_Employed** so I could show only those candidates. I then put **Application_Status** in Rows and for Values I used **Application_ID** (by **count**). On the PivotTable, I changed values to **% of Column Total** to get the percentage.

Self_Employed	Yes

Application Status	Percentage of Total Applications
Y	65.71%
N	34.29%



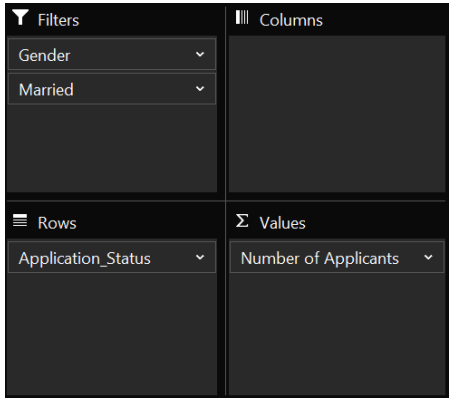
To best show this in a visual I chose to show the difference between approved and not approved with a **Pie Chart**. This chart shows that **65.71%** of **self-employed** applicants were **approved** and **34.29%** of **self-employed** applicants were **rejected**.



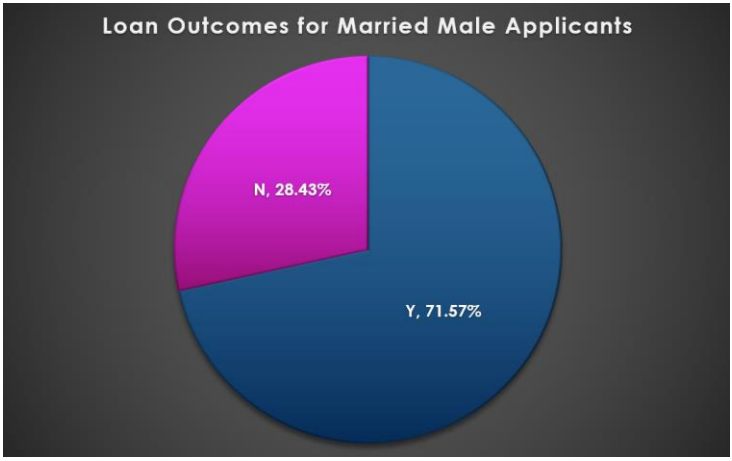
Functional Requirement 3.4

To calculate the percentage of rejection for married male applicants for loans I created a PivotTable and used **Gender** and **Married** for Filters. For Rows I used **Application_Status** and for Values I used **Number of Applicants** (by **count**). On the PivotTable, I changed values to **% of Column Total** to get the percentage.

Gender	Male
Married	Yes
Application Status	**Number of Applicants**
Y	71.57%
N	28.43%



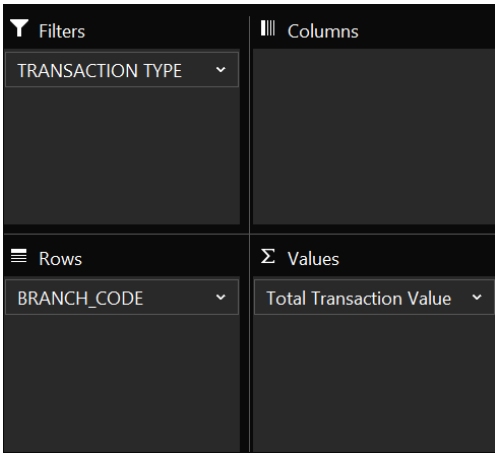
For the filters I used **Male** for **Gender** and **Yes** for **Married**. I used a **Pie Chart** to show that **28.43%** of **married male** applicants were **rejected** and **71.57%** were **approved**.



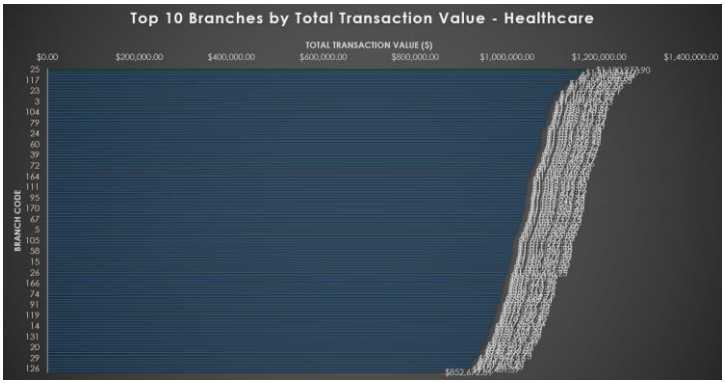
Functional Requirement 3.5

To calculate which branch processed the highest total dollar value of healthcare transactions I created PivotTable where I used **Transaction_Type** for filter so I can look only at healthcare. For Rows I used **Branch_Code** and **Values** I used the **SUM of Transaction_Amount**. I then did a sort by Largest to Smallest.

TRANSACTION TYPE	Healthcare
Branch Code	**Total Transaction Value**
25	$217,173.71
21	$205,842.23
83	$196,415.73
90	$195,492.47
60	$194,124.68
7	$190,486.60
103	$185,830.74
23	$185,809.70
109	$185,237.03
93	$179,568.89
72	$177,651.77

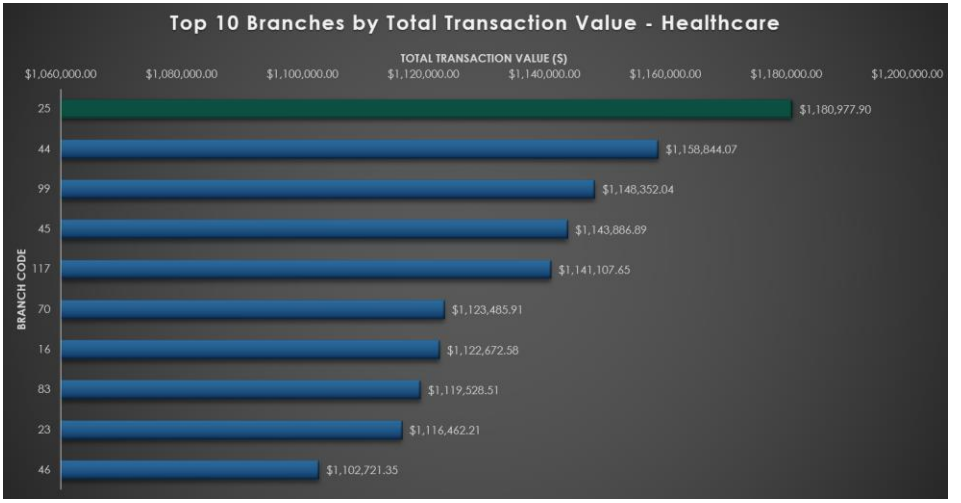


To visualize this data, I used a **Clustered Bar Chart** since it's the best way to compare branch totals for one transaction type.



Due to there being many branches, showing them all made the chart very cluttered and hard to read. Since the goal was to identify the top-performing branch, I applied a Top 10 filter, on **Branch Code**, to show only the 10 highest branches by total transaction value. This made the chart easier to understand and focused on the main insight. The results show that **Branch 25** processed the highest total dollar value for Healthcare transactions. I highlighted **Branch 25** in a different color so it stands out clearly as the top branch.

This analysis immediately identifies **Branch 25** as the key location for healthcare-related spending, making it a focal point for any further analysis in this category.



Functional Requirement 3.6

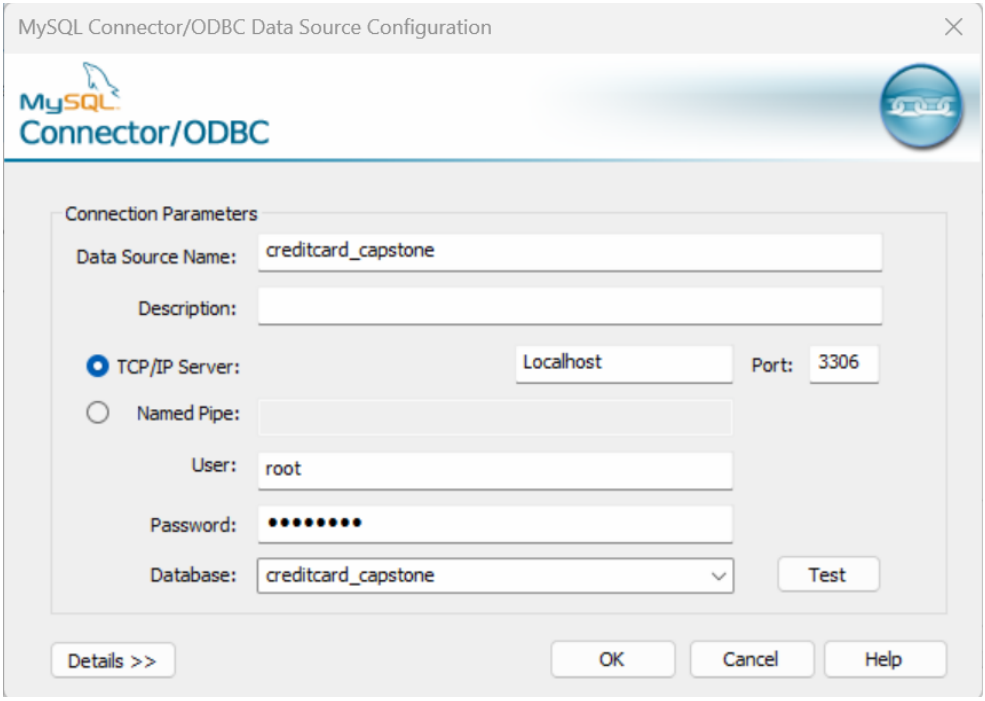
To create a query that will calculate the top three months with the largest volume of transaction data I added the following code to my **creditcard_capstone** script. Since the source data stores months as numbers, I added a **Month_Name** column in MySQL that returns the full month name for each row.

```
SELECT
   YEAR, -- extract the year of each transaction
   MONTH, -- extract the month (numeric) of each transaction
   MONTHNAME(STR_TO_DATE(CONCAT(YEAR, '-', LPAD(MONTH, 2, '0'), '-01'), '%Y-%m-%d')) AS Month_Name, -- derive a proper month name from year+month
   COUNT(TRANSACTION_ID) AS total_transactions -- count total transactions for each year+month
FROM cdw_sapp_transaction -- pulls data from the main transaction table
GROUP BY YEAR, MONTH -- group by both year and month to aggregate correctly
ORDER BY total_transactions DESC -- sort from highest to lowest
LIMIT 3; -- show top 3 months with most transactions
```

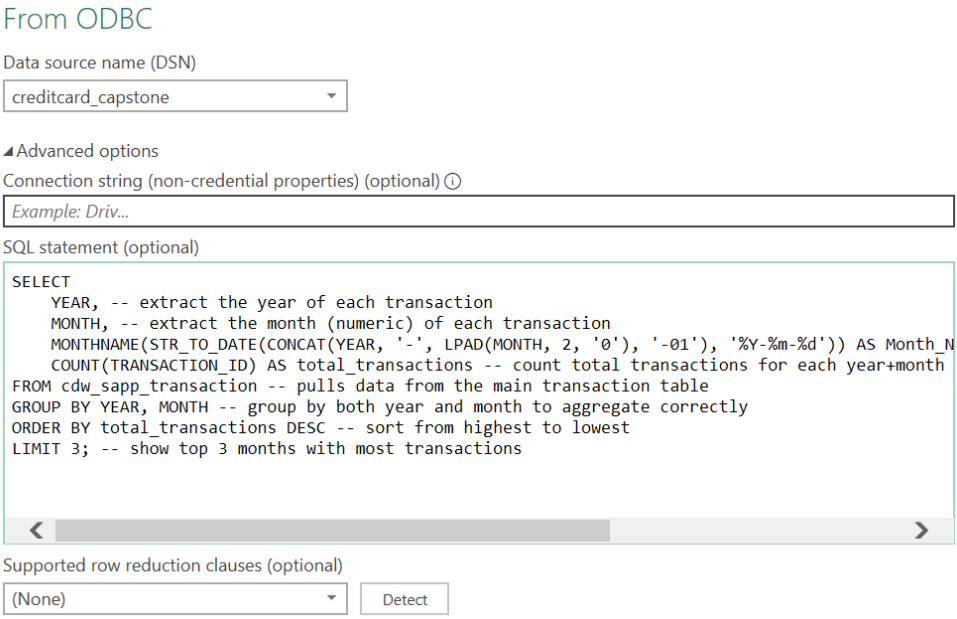
That code returned the following results in MySQL Workbench:

YEAR	MONTH	Month_Name	total_transactions
2018	2	February	3959
2018	5	May	3946
2018	10	October	3946

To create the connection between Excel and the **creditcard_capstone** database I used **ODBC** and added a new **User DSN** named "**creditcard_capstone**" with the following Parameters:



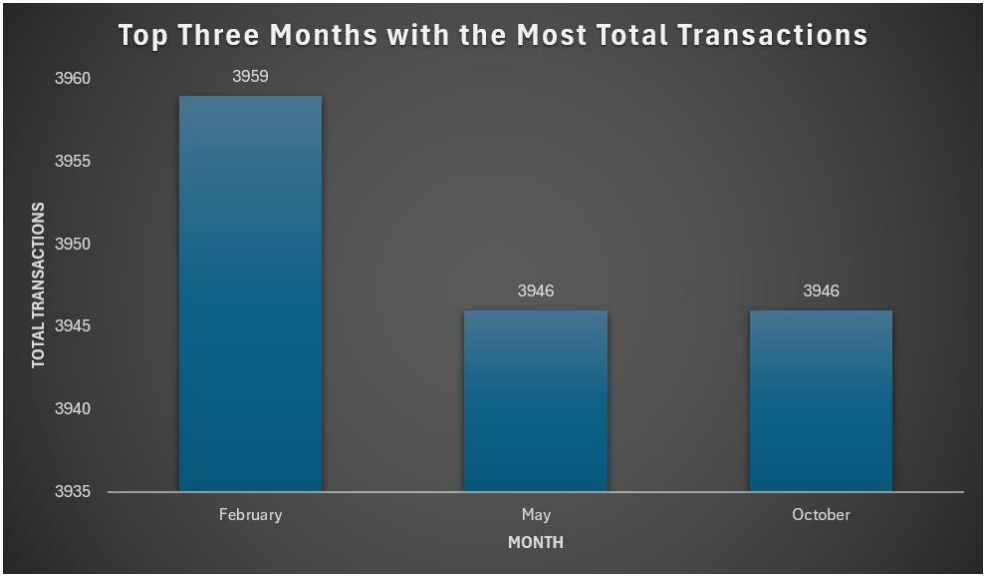
Once the connection was established I opened my Excel project and selected **Data** -> **Get Data** -> **From Other Sources** -> **From ODBC**. From there I chose the Data Source name (DSN) **creditcard_capstone**. I then expanded Advanced options and for the SQL statement I entered the code from MySQL.



This created a table in Excel with the correctly imported data which was renamed to tab FR 3.6.

YEAR	MONTH	Month_Name	total_transactions
2018	2	February	3959
2018	5	May	3946
2018	10	October	3946

To best visualize this data I chose a **Clustered Column Chart** because it clearly shows month-to-month differences in total transaction volume. This type of chart makes it easy to compare the three highest months side by side and quickly identify which period had the greatest activity.



The analysis revealed that **February**, **May**, and **October** were the months with the highest transaction volumes, suggesting potential seasonal peaks in customer activity that could be targeted for marketing.

Functional Requirement 3.7

This requirement focused on identifying which customers generated the highest total transaction values across all purchases. Using an SQL query connected to the **cdw_sapp_transaction** and **cdw_sapp_customer** tables, I calculated the total spending per customer and ranked them to display the top 10. This analysis helps highlight the company's most valuable customers and provides insight into spending concentration within the client base.

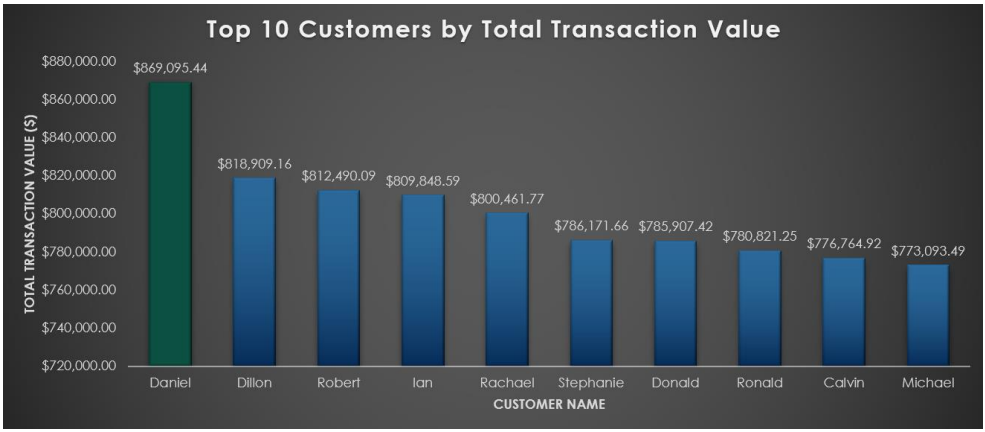
```
SELECT
   c.FIRST_NAME, -- extracts customer first name from the customers table
   c.LAST_NAME, -- extracts customer last name from the customers table
   SUM(t.TRANSACTION_AMOUNT) AS total_spent -- use the sum aggregation to
calculate total spent per customer
FROM cdw_sapp_transaction t -- pulls data from the main transaction table as the
data source with an alias of "t"
JOIN cdw_sapp_customers c -- join the customers table with alias "c"
ON t.CUST_SSN = c.SSN -- link tables on ssn
GROUP BY c.FIRST_NAME, c.LAST_NAME, t.CUST_SSN -- group by customer first name,
last name, and ssn
ORDER BY total_spent DESC -- sort the total spent largest to smallest
LIMIT 10; -- return only the top 10 customers
```

With the **ODBC** connection already created I used the above code in Excel import from **ODBC** to pull in the needed information.

That pulled all the needed data to create a table with that information. For proper formatting I changed **total_spent** to **Currency**.

FIRST_NAME	LAST_NAME	total_spent
Daniel	Meyers	$869,095.44
Dillon	Whitfield	$818,909.16
Robert	Lilly	$812,490.09
Ian	Arias	$809,848.59
Rachael	Boyce	$800,461.77
Stephanie	Snow	$786,171.66
Donald	Stone	$785,907.42
Ronald	Mathews	$780,821.25
Calvin	Hampton	$776,764.92
Michael	Dunham	$773,093.49

To show this information in visual form, I created a **Clustered Column Chart** arranged in descending order, with the **top customer** highlighted in a different color. This chart type makes it easy to compare total spending across all top customers side by side and quickly identify which individuals contribute the highest transaction values.



Functional Requirement 3.8

This requirement focused on determining how many transactions in the dataset were identified as **fraudulent**. The goal was to measure the overall scale of fraudulent activity within the company's transaction history and provide a clear count that could be compared against total transaction volume. To calculate how many transactions were flagged as fraudulent, I used the following code in MySQL:

```
SELECT
   COUNT(*) AS total_fraudulent_transactions -- counts the total number of fraud
cases
FROM cdw_sapp_transaction -- pulls data from the main transaction table
WHERE FRAUDULENT = 1; -- only include rows that are marked as fraudulent (1)
```

This returned the result of **2363** transactions were **flagged** as **fraudulent**:

	total_fraudulent_transactions
▶	2363

Using the code I again imported the data into Excel through **ODBC**.

To create a visual I need more information so I created the following code in MySQL so it would have the total number of fraudulent cases vs non-fraudulent:

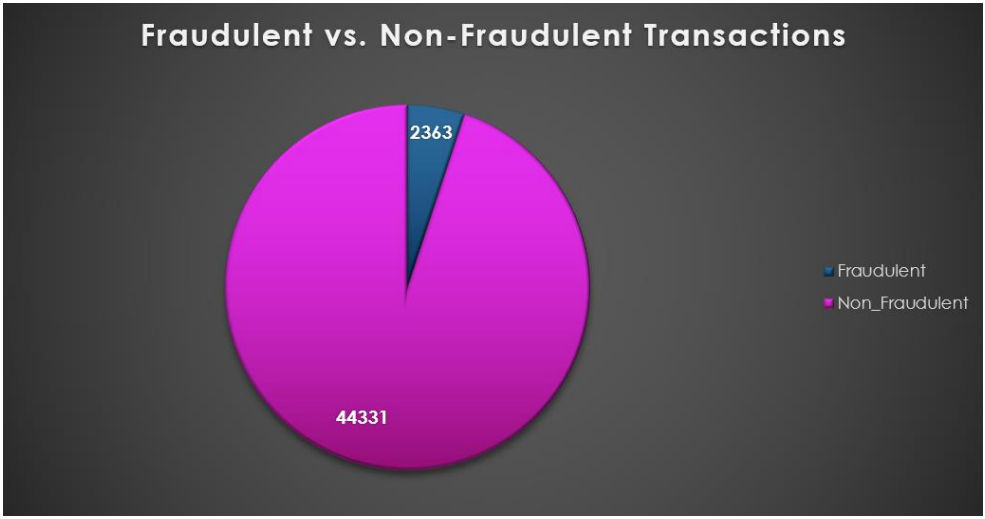
```
SELECT
   SUM(CASE WHEN FRAUDULENT = 1 THEN 1 ELSE 0 END) AS Fraudulent, -- count
how many rows have fraudulent = 1 (fraud transactions)
   SUM(CASE WHEN FRAUDULENT = 0 THEN 1 ELSE 0 END) AS Non_Fraudulent --
count how many rows have fraudulent = 0 (non-fraud transactions)
FROM cdw_sapp_transaction; -- pulls data from the main transaction table
```

I then again imported the new data into Excel through **ODBC**.

That pulled in the correct data to show the total number of Fraudulent cases and Non_Fraudulent cases:

Fraudulent	Non_Fraudulent
2363	44331

The best way to display these numbers in a visual is with a **Pie Chart** as it's directly comparing two numbers.



Functional Requirement 3.9

This requirement focused on finding what percentage of all transactions were identified as fraudulent compared to the total transaction count. The goal was to quantify the overall fraud rate in the dataset and present both fraudulent and non-fraudulent percentages for clear visual comparison. To perform this calculation, I used the following MySQL query, which divides the number of fraudulent transactions by the total count to determine the fraud rate and subtracts that value from 1 to calculate the non-fraudulent rate.

```
SELECT
  ROUND(SUM(CASE WHEN FRAUDULENT = 1 THEN 1 ELSE 0 END) / NULLIF(COUNT(*), 0), 4) AS Fraudulent_Rate, -- percent of all transactions that are fraud; NULLIF(COUNT(*),0) prevents divide by zero
  ROUND(1 - (SUM(CASE WHEN FRAUDULENT = 1 THEN 1 ELSE 0 END) / NULLIF(COUNT(*), 0)), 4) AS Non_Fraudulent_Rate -- percent of all transactions that are not fraud; NULLIF(COUNT(*),0) prevents divide by zero
FROM cdw_sapp_transaction; -- use data from transaction table
```

That output **0.0506** and **0.9494** which I'm keeping in decimal format for easier formatting in Excel.

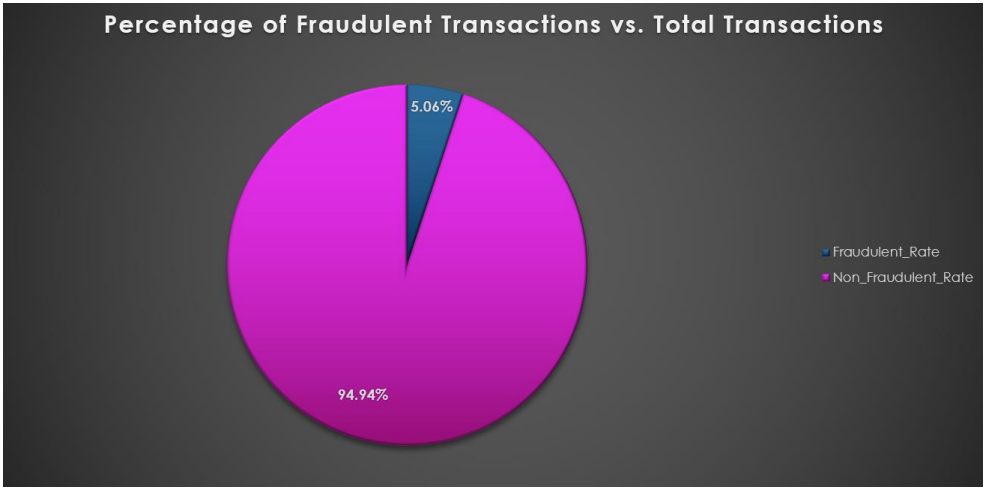
Fraudulent_Rate	Non_Fraudulent_Rate
0.0506	0.9494

I then imported the data into Excel through **ODBC**.

Once the data was loaded into Excel I changed the format to **percentage** to show the actual percentage of fraudulent transactions.

Fraudulent_Rate	Non_Fraudulent_Rate
5.06%	94.94%

To visualize this data I create a **Pie Chart** as it's directly comparing two percentages.


Percentage of Fraudulent Transactions vs. Total Transactions
5.06%
94.94%
Fraudulent_Rate
Non_Fraudulent_Rate

Functional Requirement 3.10

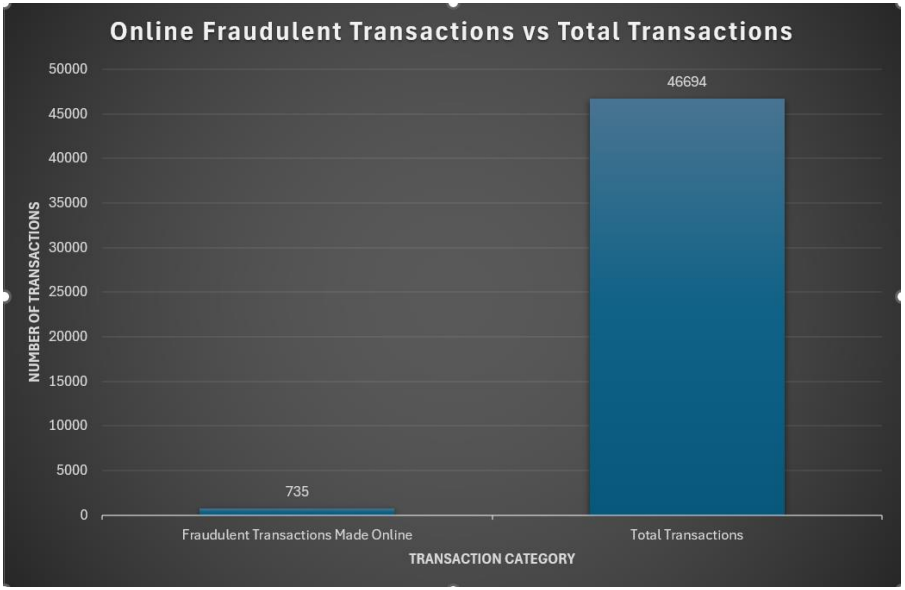
This requirement aimed to identify how many fraudulent transactions were made through online channels compared to the overall transaction count. The goal was to understand the role that online activity played in total fraud cases and provide data that could be visualized for comparison. To calculate these values, I used the following MySQL query to return both the number of online fraudulent transactions and the total number of transactions for chart creation in Excel.

```
SELECT
   COUNT(*) AS Online_Fraudulent_Count, -- total transactions that are both online and fraudulent
   (SELECT COUNT(*) FROM cdw_sapp_transaction) AS Total_Transactions -- grand total of all transactions
FROM cdw_sapp_transaction -- pulls data from the main transaction table
WHERE IS_ONLINE = 1 AND FRAUDULENT = 1; -- filter for online + fraudulent
```

Online_Fraudulent_Count	Total_Transactions
735	46694

I then loaded the data into Excel from **ODBC**.

Next, I created a **Clustered Column Chart** to compare the number of online fraudulent transactions to the total number of transactions. This chart type was chosen because it visually contrasts the two categories side by side, making it easy to see how much smaller the online fraud count is relative to the overall total.



The analysis showed that while there were **46,694** total transactions, only **735** of the **fraudulent** transactions occurred **online**. This indicates that the majority of fraudulent activity is happening through offline channels.

Functional Requirement 3.11

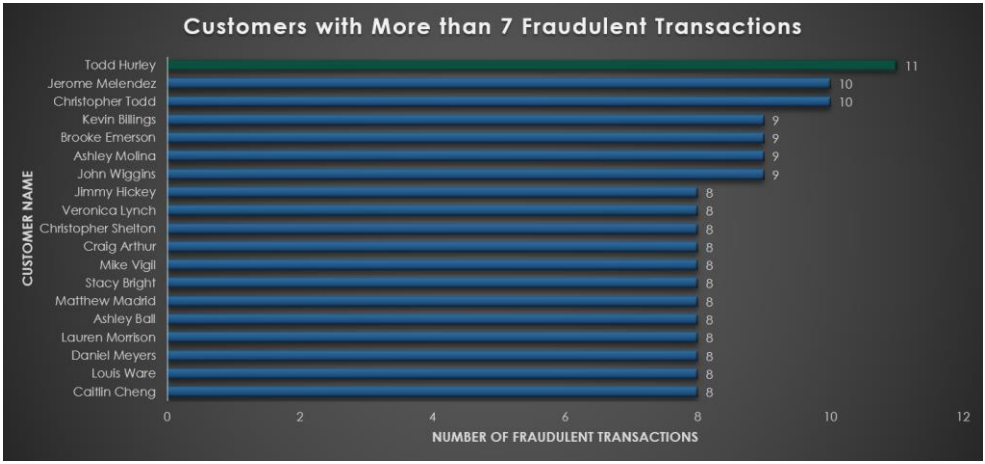
This requirement focused on identifying which customers were involved in the greatest number of fraudulent transactions. The goal was to pinpoint individuals who exceeded the threshold of seven fraudulent transactions, helping to highlight repeat offenders and potential high-risk accounts. To perform this analysis, I used the following MySQL query to calculate and list all customers meeting that criterion.

```
SELECT CONCAT(c.FIRST_NAME, ' ', c.LAST_NAME) AS Customer_Name, -- combine the customer's first name and last name into one name (this will be so the chart will show the full name)
    COUNT(t.TRANSACTION_ID) AS Fraudulent_Count -- total number of fraudulent transactions per customer
FROM cdw_sapp_transaction t -- pulls data from the main transaction table with alias "t"
JOIN cdw_sapp_customers c -- join to customer table, with alias "c" for name lookup
        ON t.CUST_SSN = c.SSN -- match customer ssn between tables
WHERE t.FRAUDULENT = 1 -- include only transactions marked as fraudulent
GROUP BY t.CUST_SSN, c.FIRST_NAME, c.LAST_NAME -- group by each customer's SSN, first name, and last name
HAVING COUNT(t.TRANSACTION_ID) > 7 -- show only those with more than 7 fraudulent transactions
ORDER BY Fraudulent_Count DESC; -- rank results from most to least fraudulent activity
```

FIRST_NAME	LAST_NAME	Fraudulent_Count
Todd	Hurley	11
Christopher	Todd	10
Jerome	Melendez	10
John	Wiggins	9
Ashley	Molina	9
Brooke	Emerson	9
Kevin	Billings	9
Caitlin	Cheng	8
Louis	Ware	8

I then imported into Excel using **ODBC**.

Next, I created a **Clustered Bar Chart** to display the customers with more than seven fraudulent transactions. This chart type was selected because it allows for easier reading of customer names along the vertical axis while clearly comparing the number of fraudulent transactions across individuals.



This showed the **Todd Hurley** was the customer with the most fraudulent transactions with **11**. This list provides a direct, actionable starting point for the fraud investigation team, allowing them to prioritize these high-frequency accounts."

Functional Requirement 4.1

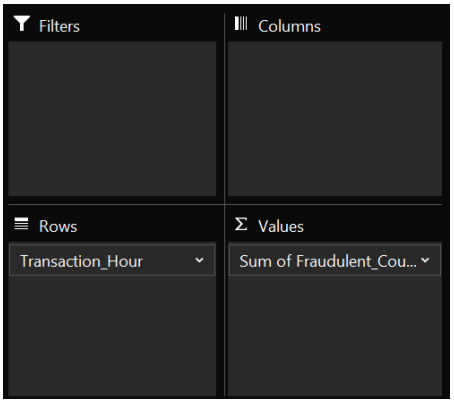
This requirement aimed to determine the specific time of day when fraudulent transactions occurred most frequently. Identifying peak fraud hours helps reveal behavioral patterns and can guide monitoring or prevention strategies. To perform this analysis, I used the following code to extract the **hour** from each **transaction** and calculate the total number of fraud cases within each time range.

```
SELECT
    HOUR(Time) AS Transaction_Hour, -- extract the hour (0–23) from the Time column
    COUNT(*) AS Fraudulent_Count -- count how many fraudulent transactions
occurred in each hour
FROM cdw_sapp_transaction -- pulls data from the main transaction table
WHERE Fraudulent = 1 -- include only fraudulent transactions
GROUP BY HOUR(Time) -- group results by each hour of the day
ORDER BY Fraudulent_Count DESC; -- rank hours by number of frauds, highest first
```

Transaction_Hour	Fraudulent_Count
6	119
2	119
22	110
16	104
19	103
17	102
0	102
4	101
20	100

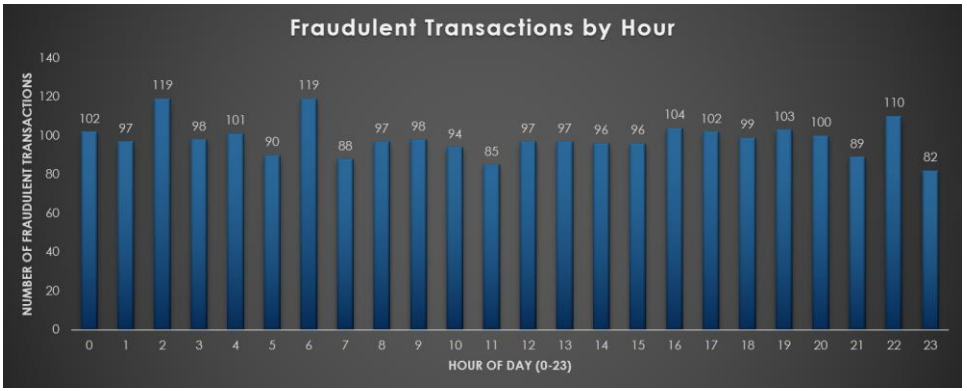
I then loaded the data into Excel through ODBC.

Using the imported data, I created a PivotTable placing **Transaction_Hour** in Rows and **Fraudulent_Count** in Values.



Transaction Hour	Sum of Fraudulent_Count
0	102
1	97
2	119
3	98
4	101
5	90
6	119
7	88
8	97
9	98
10	94
11	85
12	97
13	97
14	96
15	96
16	104
17	102
18	99
19	103
20	100
21	89
22	110
23	82

To best visualize this data, I created a **Clustered Column Chart** to display the number of fraudulent transactions by hour. This chart type was chosen because it clearly shows patterns in fraud activity throughout the day, making it easy to identify peak hours with the highest occurrence.



This shows us that **12 AM (midnight)** and **6 AM** are tied, with **119**, for the time of day when most fraudulent transactions occur.

Functional Requirement 4.2

This requirement focused on identifying which days of the week and months of the year had the highest number and percentage of fraudulent transactions. The purpose was to uncover recurring time-based patterns that could help detect fraud more effectively and improve monitoring schedules.

Step 1 – Will find the day of the week that sees the highest number of fraudulent transactions.

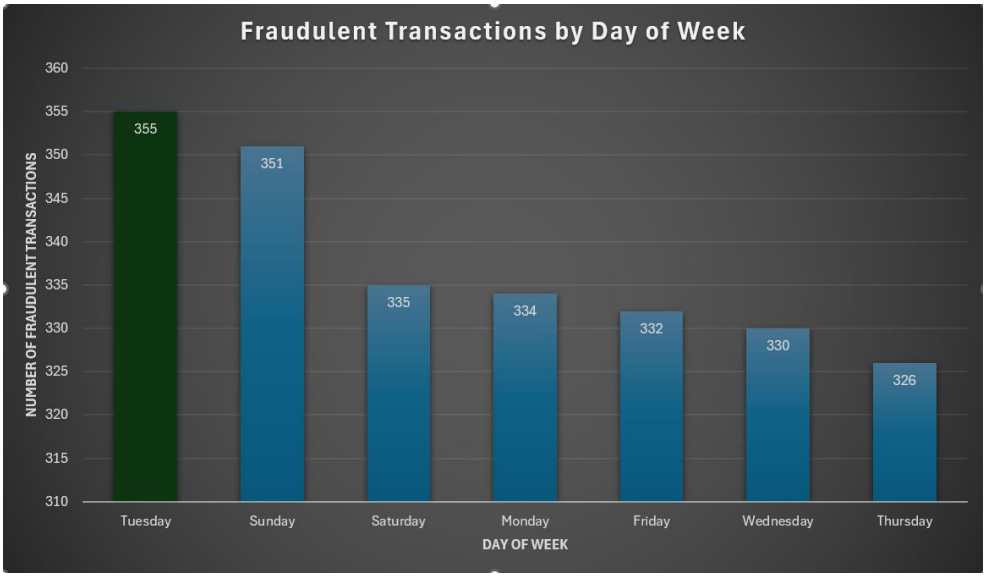
```
SELECT
    DAYNAME(CONCAT(YEAR, '-', MONTH, '-', DAY)) AS Fraud_Day, -- converts date parts into a full date and extracts weekday name
    COUNT(*) AS Fraudulent_Count -- counts how many fraudulent transactions occurred on each weekday
FROM cdw_sapp_transaction -- pulls data from the main transaction table
WHERE Fraudulent = 1 -- filters only fraudulent transactions
GROUP BY Fraud_Day -- groups results by day of week
ORDER BY Fraudulent_Count DESC; -- sorts so the most frequent day appears first
```

Fraud_Day	Fraudulent_Count
Tuesday	355
Sunday	351
Saturday	335
Monday	334
Friday	332
Wednesday	330
Thursday	326

The results show that **Tuesday** is the day with the highest number of fraudulent transactions with **355**.

I then loaded the data into Excel from **ODBC**.

To best visualize this data, I created a **Clustered Column Chart** to display the number of fraudulent transactions by day of the week. This chart type was chosen because it allows for easy comparison across days, clearly highlighting which days experience higher fraud volume.



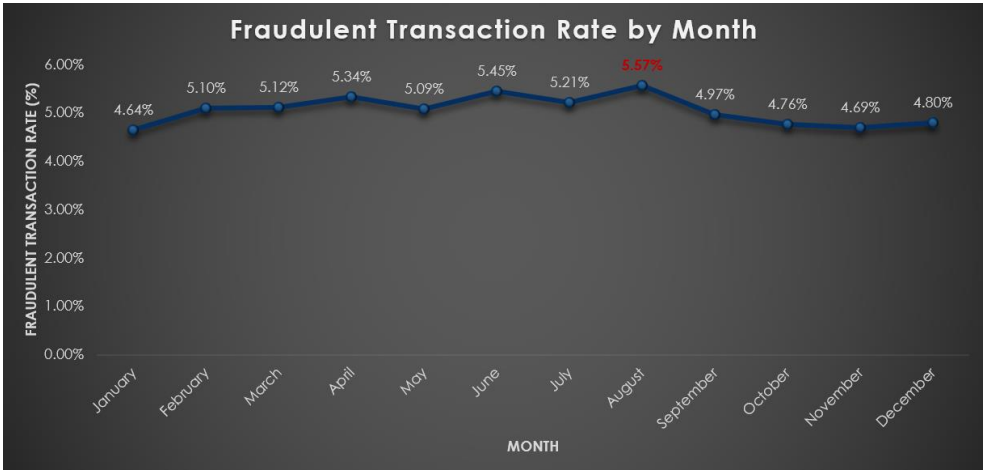
Step 2 – Find which month has the highest percentage of fraudulent transactions.

```
SELECT
    MONTHNAME(STR_TO_DATE(CONCAT('2024-', LPAD(MONTH, 2, '0'), '-01'), '%Y-%m-%d')) AS Month_Name, -- builds a full date like '2024-01-01' so monthname() recognizes it
    COUNT(CASE WHEN Fraudulent = 1 THEN 1 END) AS Fraudulent_Count, -- counts only fraudulent transactions
    COUNT(*) AS Total_Transactions, -- total transactions for that month
  ROUND((COUNT(CASE WHEN Fraudulent = 1 THEN 1 END) / COUNT(*)), 4) AS Fraudulent_Percentage -- calculates % fraud per month
FROM cdw_sapp_transaction -- pulls data from the main transaction table
GROUP BY MONTH, Month_Name -- groups by month name
ORDER BY CAST(MONTH AS UNSIGNED); -- ensures months appear jan–dec in correct order
```

Month_Name	Fraudulent_Count	Total_Transactions	Fraudulent_Percentage
January	177	3814	0.0464
February	202	3959	0.0510
March	197	3851	0.0512
April	205	3840	0.0534
May	201	3946	0.0509
June	208	3819	0.0545
July	205	3935	0.0521
August	215	3861	0.0557
September	193	3886	0.0497

To visualize this data in a chart I loaded the data into Excel from **ODBC**.

Once the data was loaded, I used it to create a **Line Chart** which was chosen because it effectively shows how fraud percentages change over time, making it easy to identify which month experienced the highest fraud rate. As we only want to show the month with the highest percentage of fraudulent transactions I used **Month_Name** and **Fraudulent_Percentage** in the chart.



This shows us that the month with the highest percentage of fraudulent transactions is **August** with **5.57%**.

Functional Requirement 4.3

For this requirement, the goal was to analyze and compare the average transaction amounts for fraudulent versus non-fraudulent transactions.

To accomplish this, I first wrote a SQL query to calculate the average transaction amount for each of the two groups. The query uses a **CASE** statement to label each transaction as either **'Fraudulent'** or **'Non-Fraudulent**' and then groups the results to calculate the average for each category.

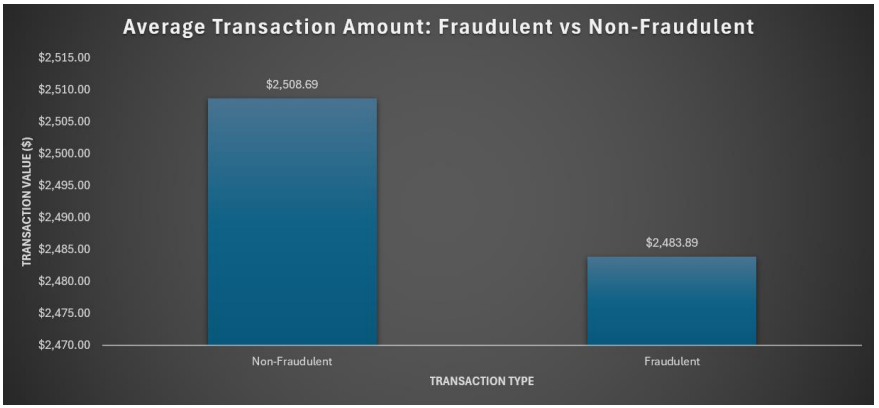
```
SELECT
   Transaction_Type, -- the fraud label created in the subquery ('Fraudulent' or
'Non-Fraudulent')
   ROUND(AVG(TRANSACTION_AMOUNT), 2) AS Avg_Transaction_Amount --
calculates the average transaction value for each type, rounded to 2 decimals
FROM (
   SELECT
      CASE
         WHEN Fraudulent = 1 THEN 'Fraudulent' -- label transactions as 'Fraudulent'
when the fraudulent flag equals 1
         ELSE 'Non-Fraudulent' -- label all remaining transactions as 'Non-
Fraudulent'
      END AS Transaction_Type, -- creates a temporary alias for use in the main
query
      TRANSACTION_AMOUNT -- selects the amount column used for averaging
   FROM cdw_sapp_transaction -- pulls data from the main transaction table
   WHERE TRANSACTION_AMOUNT IS NOT NULL -- filters out any null or missing
values that could affect the average
) AS sub -- creates a subquery (temporary dataset) named 'sub' for cleaner
grouping
GROUP BY Transaction_Type; -- groups by the fraud label to return one row per
type
```

After running that code the following results are returned.

Transaction_Type	Avg_Transaction_Amount
Non-Fraudulent	2508.69
Fraudulent	2483.89

I then loaded the data into Excel from **ODBC**.

Once loaded, I used the data to create a **Clustered Column Chart**, as it's ideal for comparing two numerical categories side by side. This makes it easy to see how the average amount of fraudulent transactions differs from non-fraudulent ones at a glance.



The chart shows that the average transaction amount for fraudulent transactions is **$2508.69** and the average non-fraudulent transaction is **$2483.89**. This is an interesting insight, as it suggests that fraudulent activities in this dataset are not necessarily targeting unusually high-value transactions but are instead blending in with normal spending patterns.

Functional Requirement 4.4

This optional task required me to determine the probability of fraud specifically for high-value transactions, defined as those over $4,000. This helps in assessing the risk associated with larger transactions.

To find this probability, I started by writing a SQL query to get two key numbers: the total count of all transactions over $4,000, and the count of how many of those were fraudulent.

```
SELECT
   COUNT(*) AS Total_Transactions_Over_4000, -- total transactions greater than $4000
   SUM(CASE WHEN Fraudulent = 1 THEN 1 ELSE 0 END) AS Fraudulent_Transactions_Over_4000 -- fraudulent subset count
FROM cdw_sapp_transaction -- pulls data from the main transaction table
WHERE TRANSACTION_AMOUNT > 4000; -- filters only high-value transactions
```

Total_Transactions_Over_4000	Fraudulent_Transactions_Over_4000
9440	490

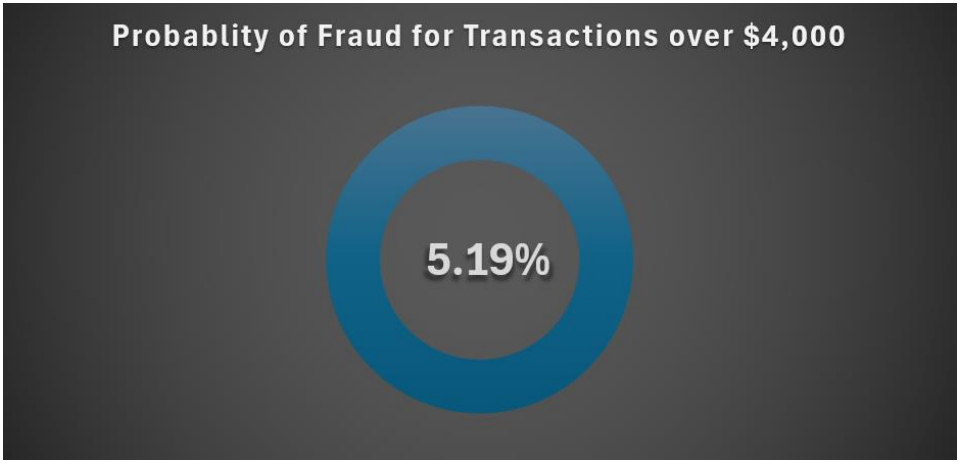
Once the query returned the two necessary values, I imported the data into Excel using the established ODBC connection. In Excel, I performed a simple division calculation:

```
=[@[Fraudulent_Transactions_Over_4000]]/[@[Total_Transactions_Over_4000]]
```

Total_Transactions_Over_4000	Fraudulent_Transactions_Over_4000	Fraudulent_Transactions_Over_4001
9440	490	5.19%

The result showed that the probability of a transaction over $4,000 being **fraudulent** is **5.19%**.

To visualize this single, important metric in a clear and impactful way, I used a **Donut Chart**. This chart is excellent for displaying a single percentage, as it focuses the viewer's attention directly on the key finding.



Functional Requirement 5.1

For my dashboard, I'm going to have a header and four KPI's to show important metrics. Those KPI will be **Total Transactions**, **Total Customers**, **Fraudulent Transactions %**, and **Approved Loan %**.

1. Total Transactions

What it shows:
This measures the overall activity level within the company's credit card operations. Every purchase, payment, or transfer processed.

Why it's important:
It reflects the scale of business operations and transaction volume. High transaction totals indicate strong customer engagement and healthy system usage. It also serves as the foundation for detecting unusual activity patterns when analyzing fraud or revenue performance.

2. Total Customers

What it shows:
This represents the total number of active customers in the system , those who are holding credit cards or accounts with the company.

Why it's important:
It provides context for all other metrics. Knowing how many customers are served helps interpret the size of the transaction base and the reach of the company's financial products. It also helps normalize other KPIs (for example, frauds per customer or applications per customer).

3. Fraudulent Transactions (%)

What it shows:
This expresses what percentage of all transactions were flagged as potentially fraudulent.

Why it's important:
Fraud monitoring is a critical performance and risk metric for any financial institution. A low percentage indicates strong fraud prevention controls, while an increase signals potential vulnerabilities. It connects directly to the project's fraud detection goal and shows how effectively the company manages risk exposure.

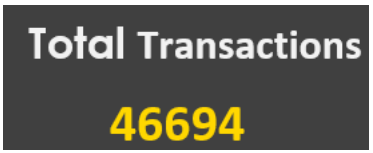
4. Approved Loan (%)

What it shows:
This measures how many loan applications were approved compared to total applications submitted.

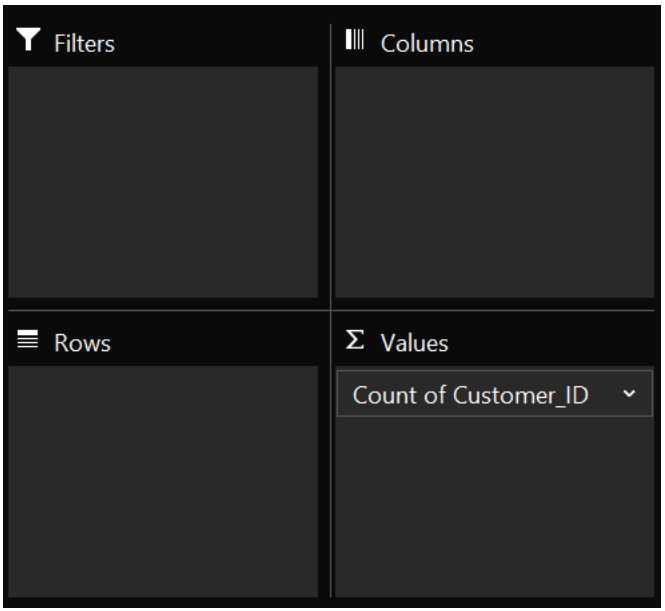
Why it's important:
It reflects the company's lending strategy and customer credit health. A balanced approval rate indicates responsible lending. Not too strict (which could limit business growth) and not too lenient (which could increase default risk). It also ties into the project's loan insights objective, helping leadership assess application success and customer eligibility trends.

To calculate **Total Transactions** I had to use the code, **='FR 3.1'!B9**, to grab the total transactions from my previous calculations on FR 3.1.

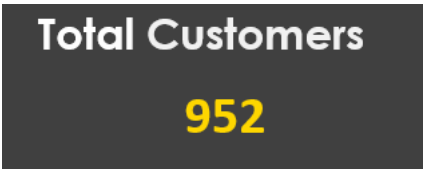
Transaction Type	Number of Transactions
Bills	6861
Healthcare	6723
Test	6683
Education	6638
Entertainment	6635
Gas	6605
Grocery	6549
Grand Total Transactions	**46694**



To calculate the **Total Customers** I had to create a new worksheet called **KPI Calculations** with a PivotTable to **Count of Customer_ID**.

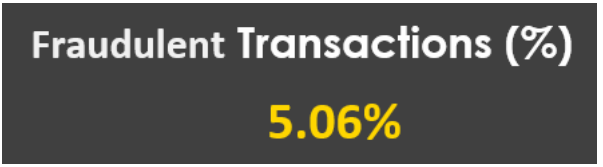


Count of Customer_ID
952



To calculate **Fraudulent Transactions (%)** I had to use the code, **='FR 3.9'!A2**, to grab the total transactions from my previous calculations on FR 3.9.

Fraudulent_Rate
5.06%

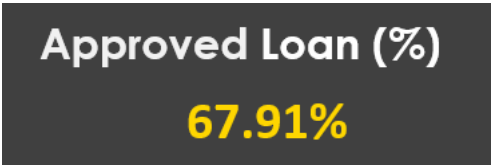


To calculate the **Approved Loan (%)** I used **KPI Calculations** with a PivotTable to pull in **Application_Status** and **Count of Customer_ID** from the **loan_application** table. This shows how many total customers applied for loans which can be divided by the total number of customers that submitted applications.

```
=C3/C4
```

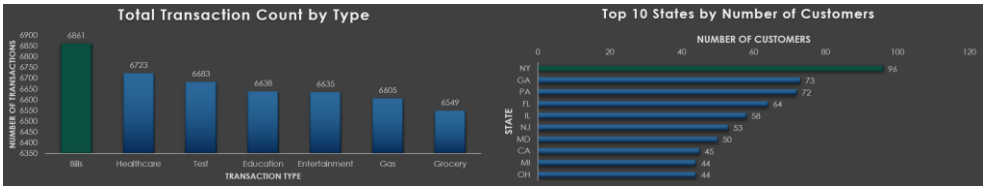
	B	C
1	Application_Status	Total Customers That Applied for Loans
2	N	164
3	Y	347
4	Grand Total	511

Percent of Approved Loans
67.91%

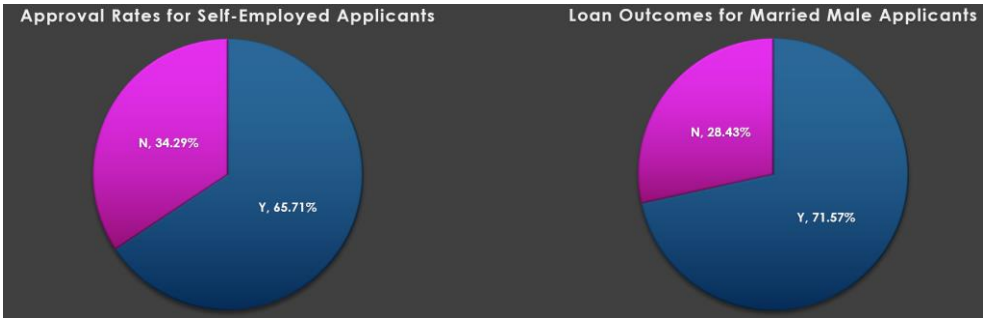


To create the rest of the dashboard I copied the charts from my functional requirements (FR 3.1 to FR 3.11) and pasted them into Dashbaord. I then removed the fill color from the charts so they blend perfectly with the background.

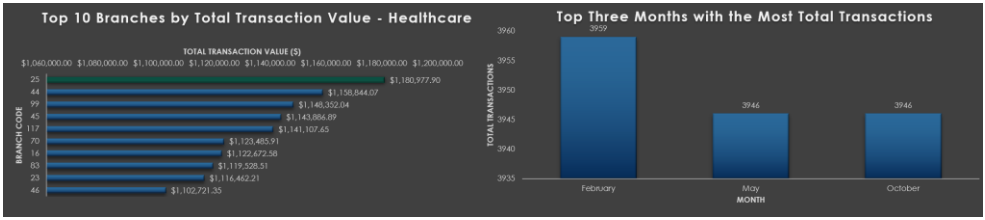
FR 3.1 and FR 3.2



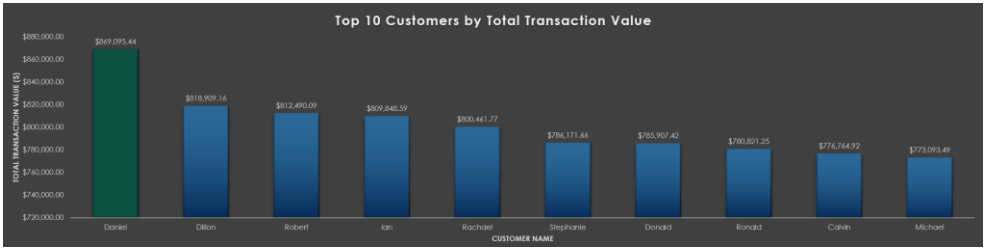
FR 3.3 and FR 3.4



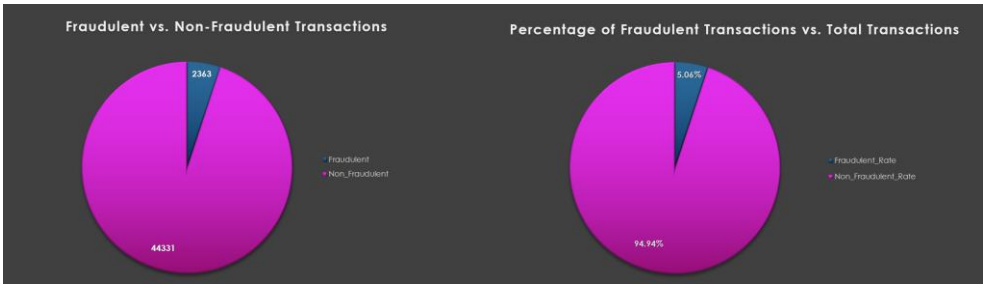
FR 3.5 and FR 3.6



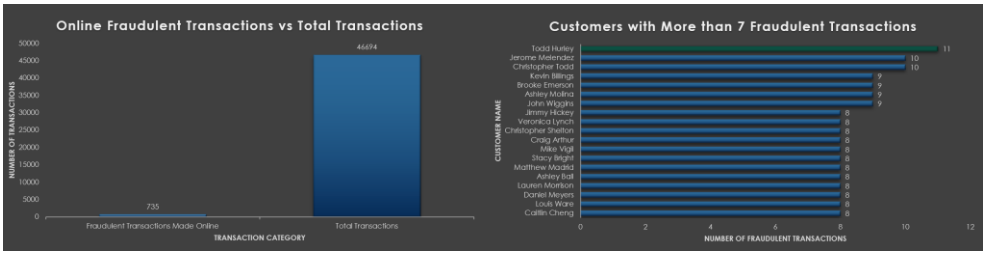
FR 3.7



FR 3.8 and FR 3.9

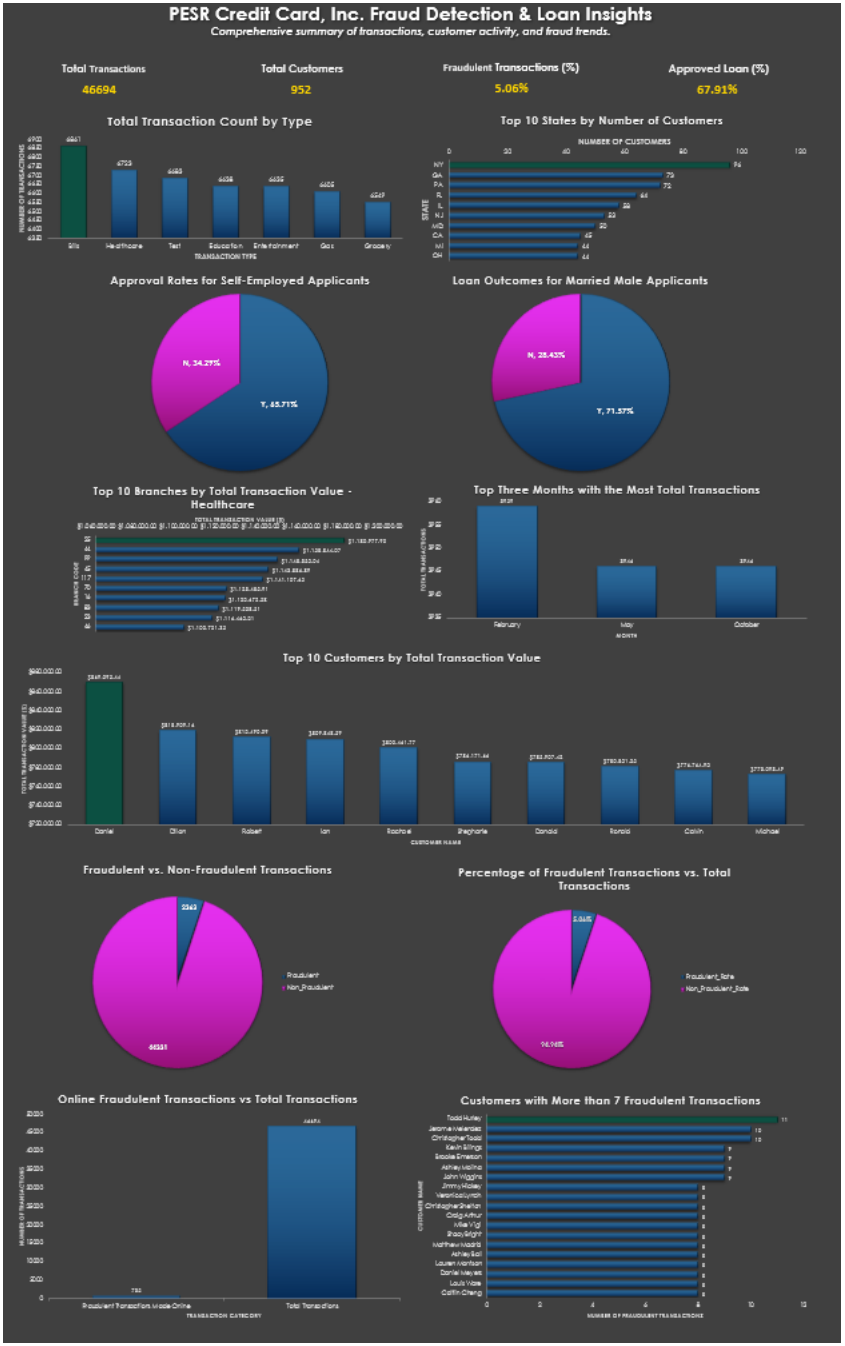


FR 3.10 and FR 3.11

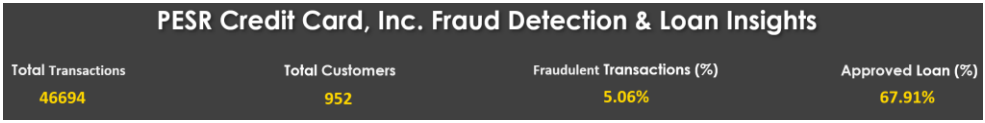


While slicers could be added for interactivity, they were intentionally left out since the Part 1 requirements focus on accurate analysis and clean presentation rather than dynamic filtering. In addition, the slicers would have affected very few elements, so including them would have been more of a distraction than a benefit.

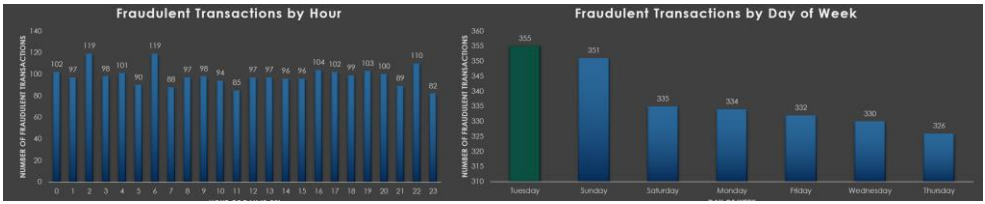
With that, the dashboard is complete.


PESR Credit Card, Inc. Fraud Detection & Loan Insights
Comprehensive summary of transactions, customer activity, and fraud trends.
Total Transactions
46694
Total Customers
952
Fraudulent Transactions (%)
5.06%
Approved Loan (%)
67.91%
Total Transaction Count by Type
Bills
Healthcare
Test
Education
Entertainment
Gas
Grocery
TRANSACTION TYPE
Top 10 States by Number of Customers
NUMBER OF CUSTOMERS
STATE
Approval Rates for Self-Employed Applicants
N, 34.29%
Y, 65.71%
Loan Outcomes for Married Male Applicants
N, 28.43%
Y, 71.57%
Top 10 Branches by Total Transaction Value - Healthcare
Top Three Months with the Most Total Transactions
February
May
October
MONTH
Top 10 Customers by Total Transaction Value
Daniel
Dillon
Robert
Ian
Rachael
Stephanie
Donald
Ronald
Collin
Michael
CUSTOMER NAME
Fraudulent vs. Non-Fraudulent Transactions
Fraudulent
Non_Fraudulent
Percentage of Fraudulent Transactions vs. Total Transactions
Fraudulent_Rate
Non_Fraudulent_Rate
Online Fraudulent Transactions vs Total Transactions
Fraudulent Transactions Made Online
Total Transactions
TRANSACTION CATEGORY
Customers with More than 7 Fraudulent Transactions
Todd Hurley
Jerome Melendez
Christopher Todd
Kevin Billings
Brooke Emerson
Ashley Molina
John Wiggins
Jimmy Maxey
Veronica Lynch
Christopher Shelton
Craig Arthur
Mike Vigil
Brad Bright
Matthew Maddox
Ashley Ball
Lauren Morrison
Daniel Mayers
Louis Ware
Collin Cheng
CUSTOMER NAME
NUMBER OF FRAUDULENT TRANSACTIONS

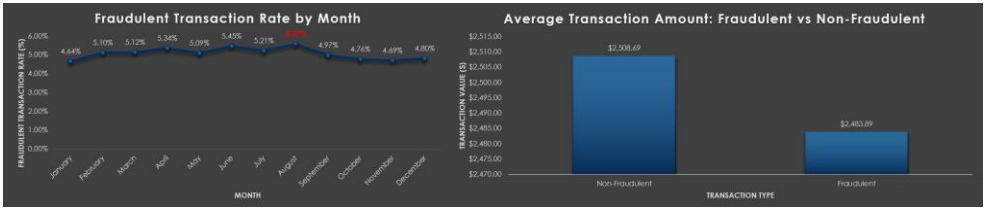
As there were optional tasks to complete, those were kept separate from the main dashboard and a second dashboard, Dashboard (Optional Analysis), was created to present the optional data. The new dashboard starts with the same heading and only switches the charts.



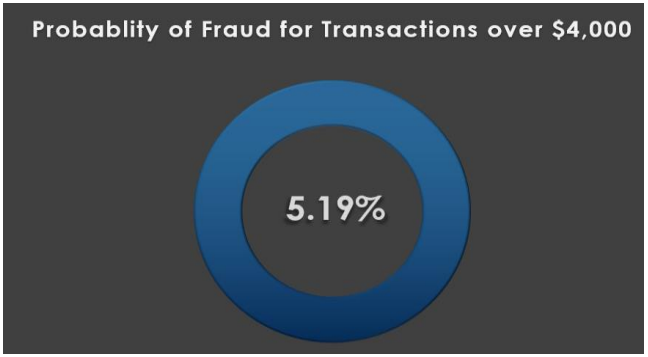
FR 4.1 and FR 4.2 Step 1



FR 4.2 Step 2 and FR 4.3



FR 4.4



With that, the dashboard is completed.