Credit Card Risk Analysis

SQL Data Validation & Profiling

Marcus Fernandes

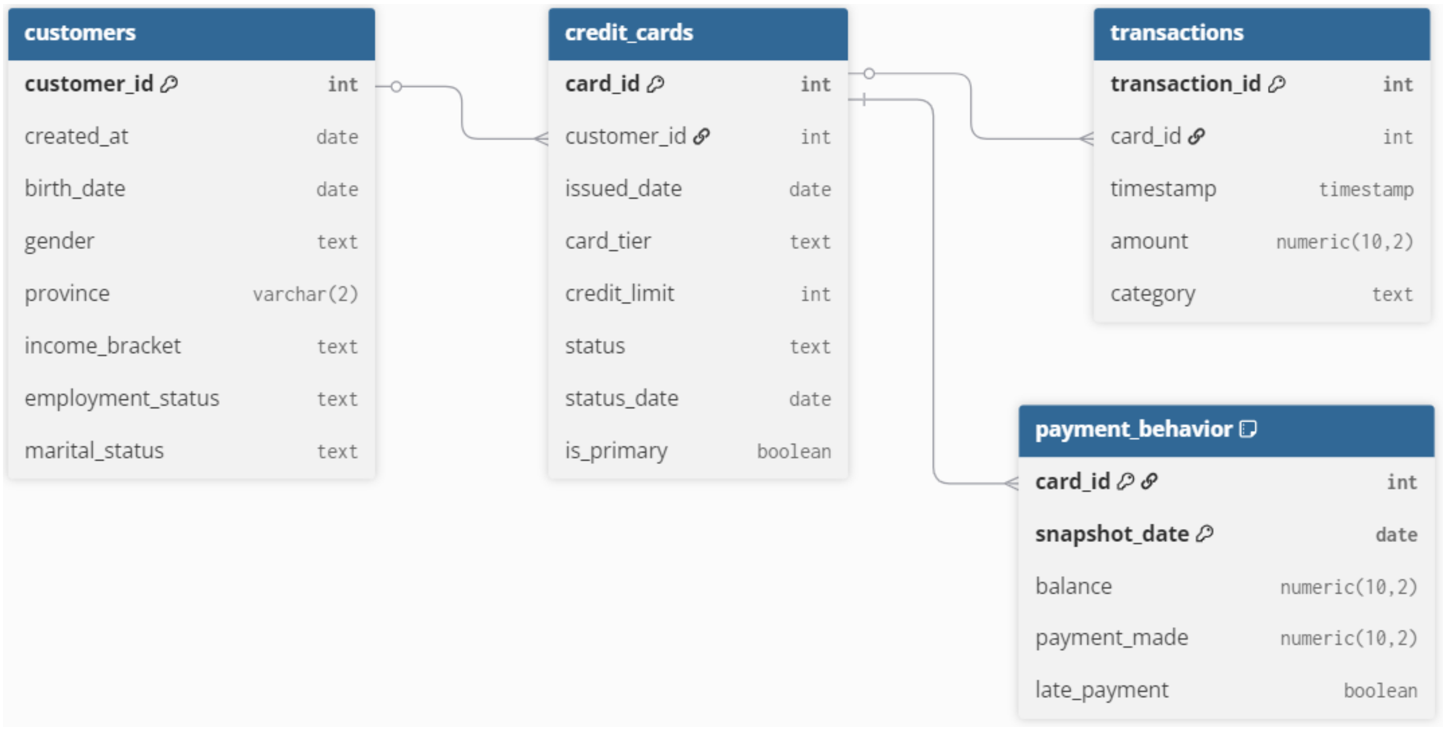
Email LinkedIn GitHub

Introduction

The Dataset is composed of four tables: customers, credit_cards, transactions, and payment_behavior. The meaning of each of them is:

Table	Meaning
customers	Identification of the customers
credit_cards	Information about their credit cards, limits and accounts
transactions	Information about their transactions and spending
payment_behavior	Information about their monthly payments and balance

Their Entity Relationship Diagram (ERD) is:



First and last names, as well as city, were excluded from the dataset to protect customer privacy. The analysis focuses solely on demographic and behavioural features. The tables have information about active clients only.

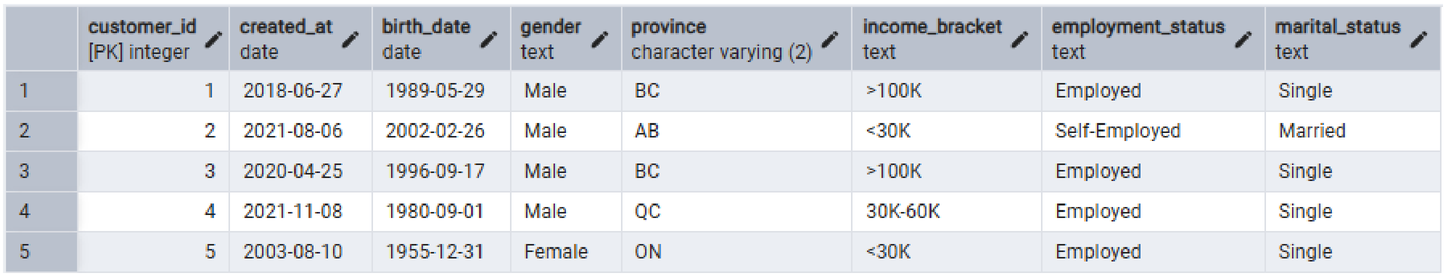
The next step is to review, clean, and prepare each table for exploration by leveraging **SQL**.

Data Cleaning and Profiling

Table: customers

1. Overview

```
-- View the first 5 rows of the dataset
SELECT * FROM customers LIMIT 5;
```



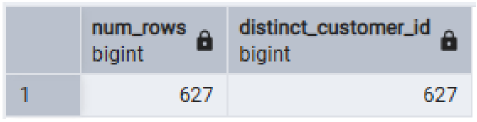
	customer_id [PK] integer	created_at date	birth_date date	gender text	province character varying (2)	income_bracket text	employment_status text	marital_status text
1	1	2018-06-27	1989-05-29	Male	BC	>100K	Employed	Single
2	2	2021-08-06	2002-02-26	Male	AB	<30K	Self-Employed	Married
3	3	2020-04-25	1996-09-17	Male	BC	>100K	Employed	Single
4	4	2021-11-08	1980-09-01	Male	QC	30K-60K	Employed	Single
5	5	2003-08-10	1955-12-31	Female	ON	<30K	Employed	Single

The table corresponds to the identification of the customers: their gender, birth date, address, income class, employment and marital status, and when they opened the account.

The table has the correct primary key (*customer_id*) and data types. It has 8 columns.

2. Duplicates and Missing Values

```
-- Check the number of rows and duplicates
SELECT
    COUNT(*) AS num_rows,
    COUNT (DISTINCT customer_id) AS distinct_customer_id
FROM customers;
```

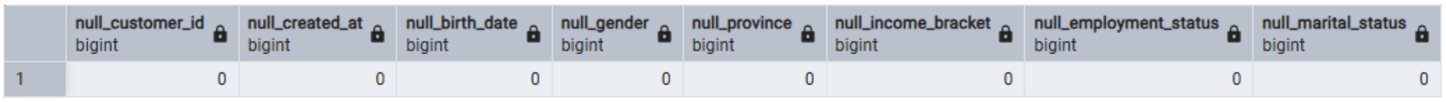


	num_rows bigint	distinct_customer_id bigint
1	627	627

There are 627 rows and the same number of distinct customer IDs, validating the Primary Key. In other words, there are 627 registered customers.

There are no duplicates, given that the number of rows and distinct IDs are the same.

```
-- Checking for missing values (NULL)
SELECT
    SUM(CASE WHEN customer_id IS NULL THEN 1 ELSE 0 END) AS null_customer_id,
    SUM(CASE WHEN created_at IS NULL THEN 1 ELSE 0 END) AS null_created_at,
    SUM(CASE WHEN birth_date IS NULL THEN 1 ELSE 0 END) AS null_birth_date,
    SUM(CASE WHEN gender IS NULL THEN 1 ELSE 0 END) AS null_gender,
    SUM(CASE WHEN province IS NULL THEN 1 ELSE 0 END) AS null_province,
    SUM(CASE WHEN income_bracket IS NULL THEN 1 ELSE 0 END) AS null_income_bracket,
    SUM(CASE WHEN employment_status IS NULL THEN 1 ELSE 0 END) AS null_employment_status,
    SUM(CASE WHEN marital_status IS NULL THEN 1 ELSE 0 END) AS null_marital_status
FROM customers;
```



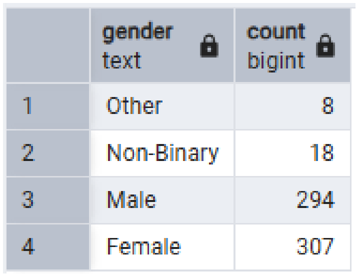
	null_customer_id bigint	null_created_at bigint	null_birth_date bigint	null_gender bigint	null_province bigint	null_income_bracket bigint	null_employment_status bigint	null_marital_status bigint
1	0	0	0	0	0	0	0	0

There are no missing values (or NULLs).

3. Column-by-Column Review

The objective is to check for unusual values and discover the unique categories on each column.

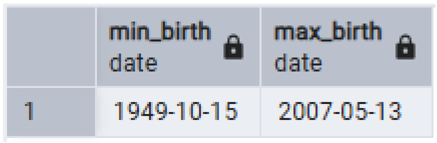
```
-- Checking for unique and unusual values in
gender
SELECT
    gender,
    COUNT(*)
FROM customers
GROUP BY 1;
```



	gender text	count bigint
1	Other	8
2	Non-Binary	18
3	Male	294
4	Female	307

Gender has four valid categories, with a similar number of males and females, and a lower number of non-binary and others.

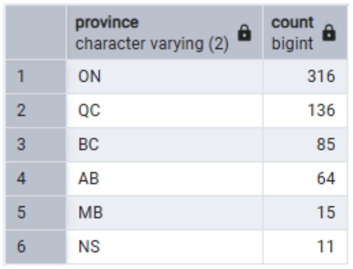
```
-- Checking for unusual values in birth_date
SELECT
    MIN(birth_date) AS min_birth,
    MAX(birth_date) AS max_birth
FROM customers;
```



	min_birth date	max_birth date
1	1949-10-15	2007-05-13

No unusual values detected in *birth_date*.

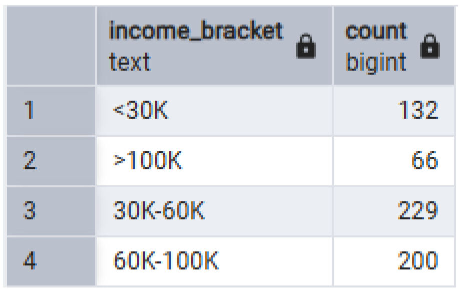
```
-- Checking for unusual values in province
SELECT
    DISTINCT province,
    COUNT(*)
FROM customers
GROUP BY 1
ORDER BY 2 DESC;
```



	province character varying (2)	count bigint
1	ON	316
2	QC	136
3	BC	85
4	AB	64
5	MB	15
6	NS	11

There are only six Provinces or Territories of Canada in this dataset, which is possible.

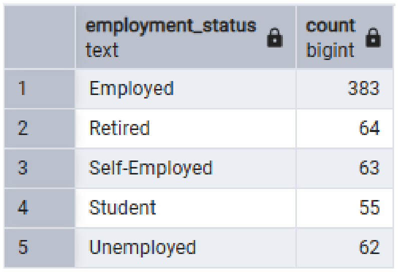
```
-- Checking for unique and unusual values in
income_bracket
SELECT
    DISTINCT (income_bracket),
    COUNT(*)
FROM customers
GROUP BY 1;
```



	income_bracket text	count bigint
1	<30K	132
2	>100K	66
3	30K-60K	229
4	60K-100K	200

There are four classes of income. Most of the customers are in the 30K-60K (229) and 60K-100K (200) brackets. The one with the least is >100K (66). No unusual values spotted.

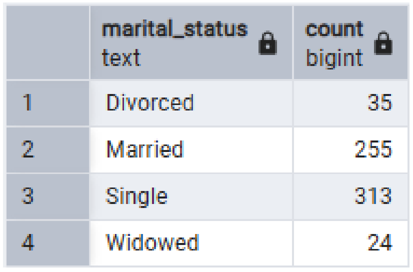
```
-- Checking for unique and unusual values in
employment_status
SELECT
    DISTINCT (employment_status),
    COUNT(*)
FROM customers
GROUP BY 1;
```



	employment_status text	count bigint
1	Employed	383
2	Retired	64
3	Self-Employed	63
4	Student	55
5	Unemployed	62

Most customers are employed (383). The remaining are similarly classified as retired, self-employed, student, and unemployed. No unusual values detected.

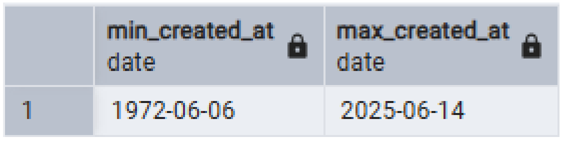
```
-- Checking for unique and unusual values in
marital_status
SELECT
    DISTINCT (marital_status),
    COUNT(*)
FROM customers
GROUP BY 1;
```



	marital_status text	count bigint
1	Divorced	35
2	Married	255
3	Single	313
4	Widowed	24

Most of the customers are single (313), followed by married ones (255). There are no unusual values.

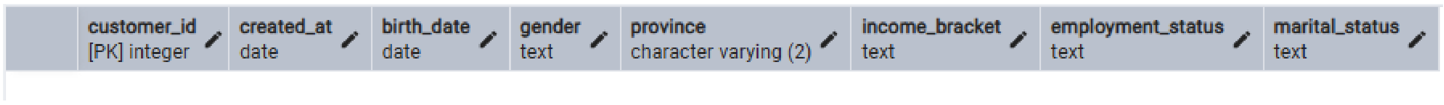
```
-- Checking for unusual values in created_at
SELECT
    MIN(created_at) AS min_created_at,
    MAX(created_at) AS max_created_at
FROM customers;
```



	min_created_at date	max_created_at date
1	1972-06-06	2025-06-14

No unusual values detected in *created_at*. There is data from 1972 to 2025.

```
-- Checking for consistency in the dates
SELECT * FROM customers WHERE birth_date > created_at OR created_at > CURRENT_DATE;
```



	customer_id [PK] integer	created_at date	birth_date date	gender text	province character varying (2)	income_bracket text	employment_status text	marital_status text

There are no consistency problems with the date columns: *created_at* never comes before *birth_date*, and *created_at* never comes after *today*.

4. Transformation and Grouping

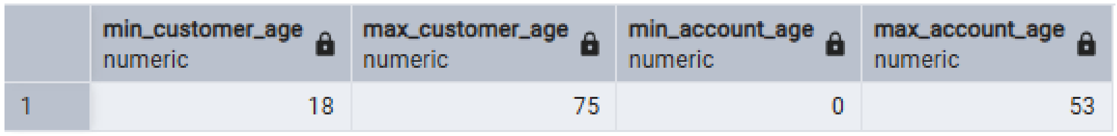
It is possible to calculate the customers' and the accounts' ages from the *birth_date* and the *created_at* variables, respectively. They can be grouped into categories and used to uncover patterns during the analysis. However, we shouldn't add columns to the *customers* table, so we can create a view and add the columns to it.

```
-- Creating VIEW with customer_age and account_age
CREATE OR REPLACE VIEW v_customers AS
SELECT
  *,
  EXTRACT(YEAR FROM AGE(CURRENT_DATE, birth_date)) AS customer_age,
  EXTRACT(YEAR FROM AGE(CURRENT_DATE, created_at)) AS account_age
FROM customers;
```

The view (*v_customers*) is created and will be used for further analysis, while keeping the customers table untouched.

Now we can check for unusual values in the age columns.

```sql
-- Checking for outliers in the age columns
SELECT
  MIN(customer_age) AS min_customer_age,
  MAX(customer_age) AS max_customer_age,
  MIN(account_age) AS min_account_age,
  MAX(account_age) AS max_account_age
FROM v_customers;
```



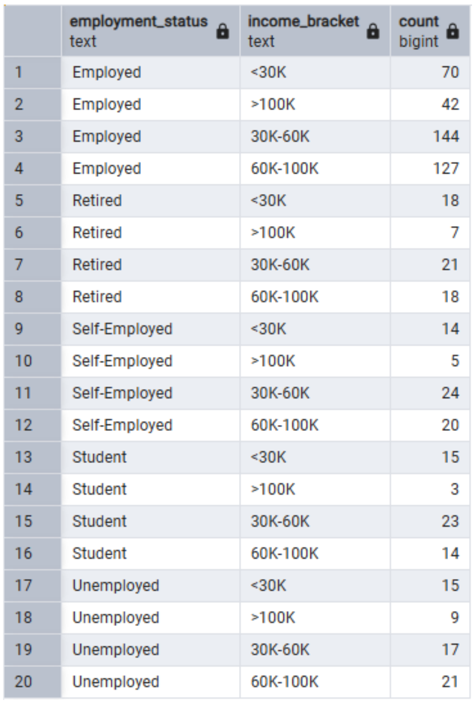
	min_customer_age numeric	max_customer_age numeric	min_account_age numeric	max_account_age numeric
1	18	75	0	53

Both the customer's and the account's ages are acceptable.

5. Cross-Tab Check

The objective is to spot obvious data quality issues.

```sql
-- Checking for cross-tab consistency
SELECT
  employment_status,
  income_bracket,
  COUNT(*)
FROM v_customers
GROUP BY 1, 2
ORDER BY 1, 2;
```



	employment_status text	income_bracket text	count bigint
1	Employed	<30K	70
2	Employed	>100K	42
3	Employed	30K-60K	144
4	Employed	60K-100K	127
5	Retired	<30K	18
6	Retired	>100K	7
7	Retired	30K-60K	21
8	Retired	60K-100K	18
9	Self-Employed	<30K	14
10	Self-Employed	>100K	5
11	Self-Employed	30K-60K	24
12	Self-Employed	60K-100K	20
13	Student	<30K	15
14	Student	>100K	3
15	Student	30K-60K	23
16	Student	60K-100K	14
17	Unemployed	<30K	15
18	Unemployed	>100K	9
19	Unemployed	30K-60K	17
20	Unemployed	60K-100K	21

Most of the table looks consistent, showing expected results for employed, retired, and self-employed profiles.

There are a few *students* and *unemployed* customers in the >100K bracket. This is unusual, but still plausible.

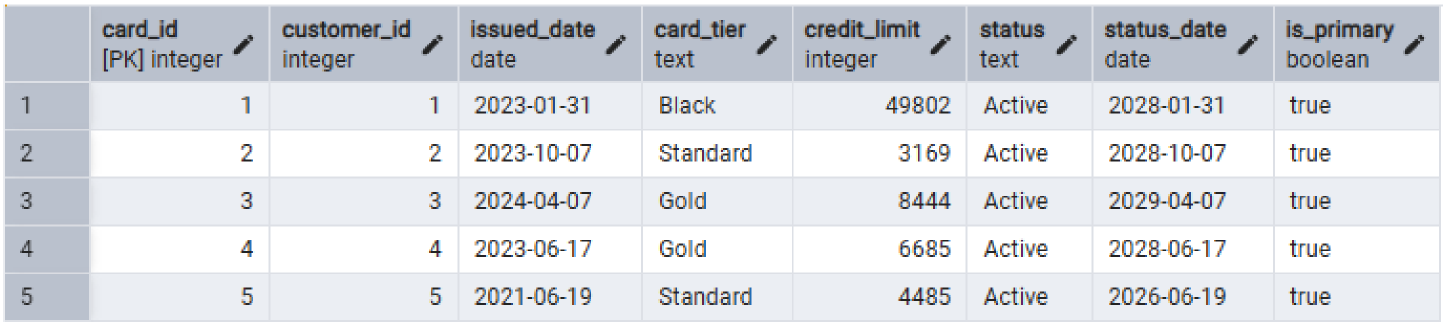
6. Conclusion

- No duplicates, missing values, unusual values, or consistency problems were identified in the dataset columns.
- A view was created with the addition of age columns.
- There are unusual customers in the >100K bracket, but it is plausible.
- We can now analyze the next table.

Table: credit_cards

From this point on, the SQL code used for the customers table will not be repeated. Only new or relevant queries specific to each table will be included.

1. Overview

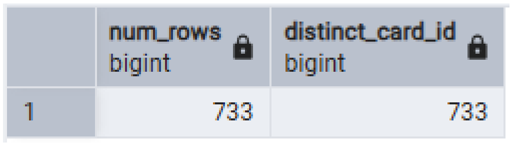


	card_id [PK] integer	customer_id integer	issued_date date	card_tier text	credit_limit integer	status text	status_date date	is_primary boolean
1	1	1	2023-01-31	Black	49802	Active	2028-01-31	true
2	2	2	2023-10-07	Standard	3169	Active	2028-10-07	true
3	3	3	2024-04-07	Gold	8444	Active	2029-04-07	true
4	4	4	2023-06-17	Gold	6685	Active	2028-06-17	true
5	5	5	2021-06-19	Standard	4485	Active	2026-06-19	true

The table provides the credit card information and status for each customer. The column *is_primary* indicates whether the card is their primary one, because a person can have multiple cards. *status_date* corresponds to the date when the status changed or will change. If the date is in the past, the *status* cannot be *Active*. If it is the future, it corresponds to the expiry date.

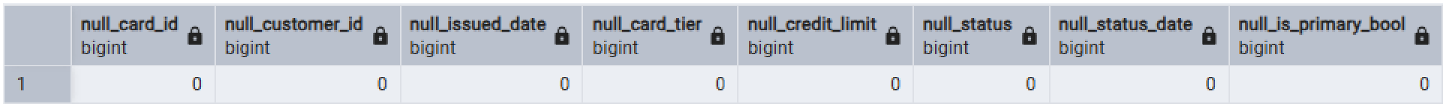
The table has the correct primary key (*card_id*) and foreign key (*customer_id*, not shown in the GUI) and contains 8 columns. The data types are correct.

2. Duplicates and Missing Values



	num_rows bigint	distinct_card_id bigint
1	733	733

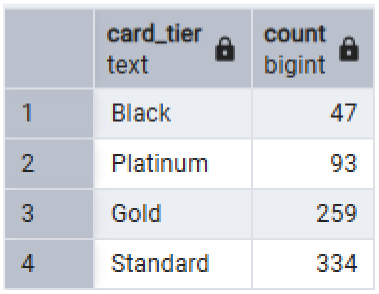
There are 733 rows and the same number of distinct card IDs, validating the Primary Key. In other words, there are 733 registered credit cards. There are no duplicates, given that the number of rows and distinct IDs are the same.



	null_card_id bigint	null_customer_id bigint	null_issued_date bigint	null_card_tier bigint	null_credit_limit bigint	null_status bigint	null_status_date bigint	null_is_primary_bool bigint
1	0	0	0	0	0	0	0	0

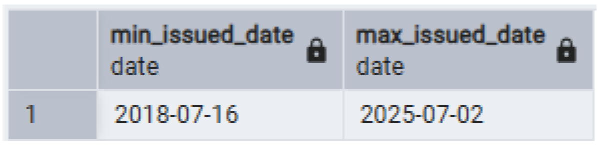
There are no missing values (or NULLs).

3. Column-by-Column Review

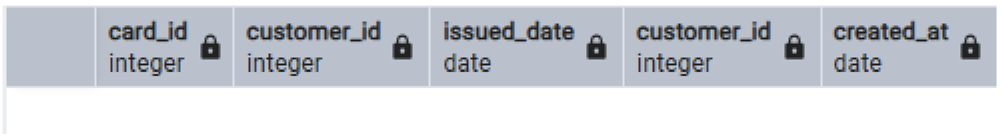


	card_tier text	count bigint
1	Black	47
2	Platinum	93
3	Gold	259
4	Standard	334

There are four card tiers, with Standard being by far the most popular one, while Black is the least popular.



	min_issued_date date	max_issued_date date
1	2018-07-16	2025-07-02

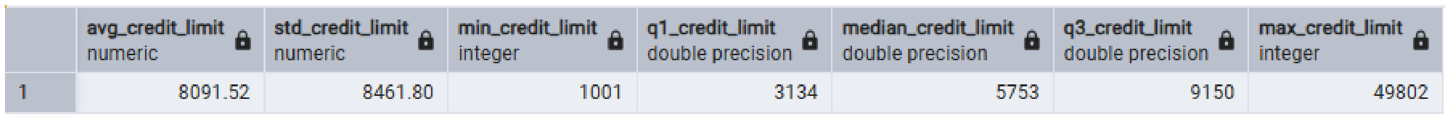


	card_id integer	customer_id integer	issued_date date	customer_id integer	created_at date

No unusual values detected in *issued_date*. The variable indicates we have recent data, with cards issued in the last 7 years.

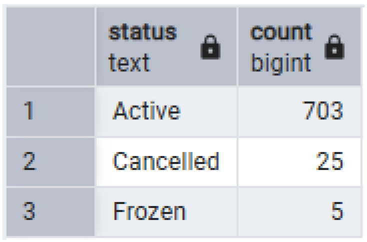
Now we can analyze *credit_limit*, which is the first numerical variable in this analysis. We can run a statistical summary to identify problems and find insights.

```
-- Basic descriptive stats for credit_limit
SELECT
  ROUND(AVG(credit_limit),2) AS avg_credit_limit,
  ROUND(STDDEV(credit_limit),2) AS std_credit_limit,
  MIN(credit_limit) AS min_credit_limit,
  PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY credit_limit) AS q1_credit_limit,
  PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY credit_limit) AS median_credit_limit,
  PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY credit_limit) AS q3_credit_limit,
  MAX(credit_limit) AS max_credit_limit
FROM credit_cards;
```



	avg_credit_limit numeric	std_credit_limit numeric	min_credit_limit integer	q1_credit_limit double precision	median_credit_limit double precision	q3_credit_limit double precision	max_credit_limit integer
1	8091.52	8461.80	1001	3134	5753	9150	49802

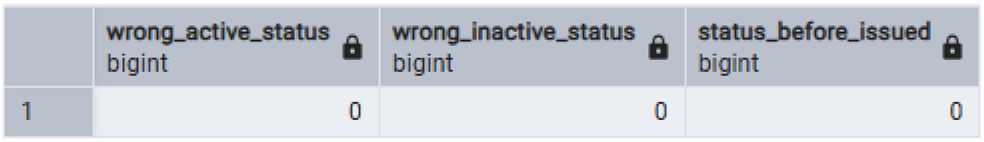
The credit card limits range from $1,001.00 to $49,802.00. The average is $8,091.52, indicating that most limits tend towards the lower range. This is confirmed by the third quartile value, which indicates that 75% of the cards have a limit under $9,150.00 (right-skewed distribution). Finally, the remaining 25% of cards have high variability, as indicated by the large standard deviation and maximum values.



	status text	count bigint
1	Active	703
2	Cancelled	25
3	Frozen	5

Most credit cards are *Active* (703), with only a few marked as *Cancelled* (25) or *Frozen* (5).

```
-- Checking for consistency in status_date
SELECT
  SUM(CASE WHEN status_date < CURRENT_DATE AND status = 'Active' THEN 1 ELSE 0 END) AS wrong_active_status,
  SUM(CASE WHEN status_date > CURRENT_DATE AND status != 'Active' THEN 1 ELSE 0 END) AS wrong_inactive_status,
  SUM(CASE WHEN status_date < issued_date THEN 1 ELSE 0 END) AS status_before_issued
FROM credit_cards;
```

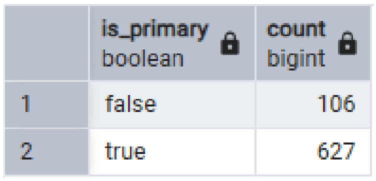


	wrong_active_status bigint	wrong_inactive_status bigint	status_before_issued bigint
1	0	0	0

wrong_active_status indicates if there has been a status change in the past, according to *status_date*, but *status* hasn't been changed to Cancelled or Frozen.

wrong_inactive_status is the opposite error, indicating if the card has the wrong *status, Cancelled* or *Frozen*, but no status change has been registered under *status_date*.

Finally, *status_before_issued* shows if the card's status changed before its issue date.
Therefore, there are no unusual values in *status_date*.



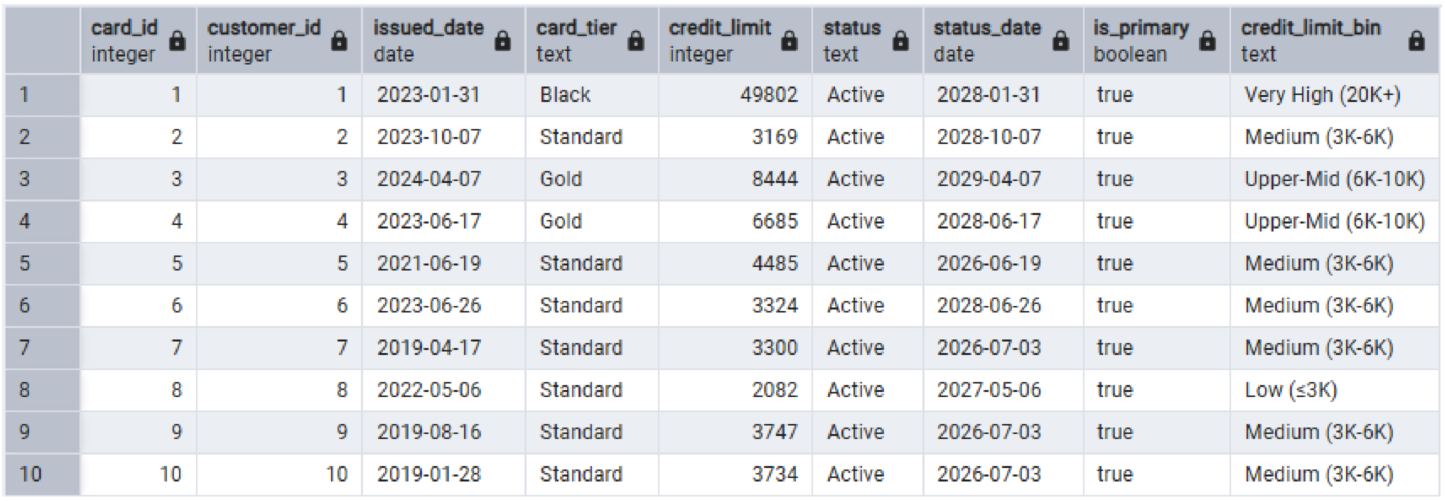
	is_primary boolean	count bigint
1	false	106
2	true	627

106 cards, or 14.5% of them, are secondary.

4. Transformation and Grouping

We can create *credit_limit* bins that will facilitate analyzing and visualizing it later. We can group them to separate low, mid, and high segments.

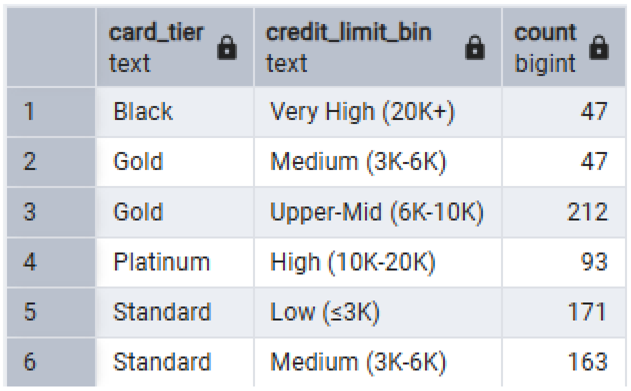
```sql
-- Creating a new VIEW with credit_limit bands
CREATE OR REPLACE VIEW v_credit_cards AS
SELECT
  *,
  CASE
    WHEN credit_limit <= 3000 THEN 'Low (≤3K)'
    WHEN credit_limit <= 6000 THEN 'Medium (3K-6K)'
    WHEN credit_limit <= 10000 THEN 'Upper-Mid (6K-10K)'
    WHEN credit_limit <= 20000 THEN 'High (10K-20K)'
    ELSE 'Very High (20K+)'
  END AS credit_limit_bin
FROM credit_cards;
```



	card_id integer	customer_id integer	issued_date date	card_tier text	credit_limit integer	status text	status_date date	is_primary boolean	credit_limit_bin text
1	1	1	2023-01-31	Black	49802	Active	2028-01-31	true	Very High (20K+)
2	2	2	2023-10-07	Standard	3169	Active	2028-10-07	true	Medium (3K-6K)
3	3	3	2024-04-07	Gold	8444	Active	2029-04-07	true	Upper-Mid (6K-10K)
4	4	4	2023-06-17	Gold	6685	Active	2028-06-17	true	Upper-Mid (6K-10K)
5	5	5	2021-06-19	Standard	4485	Active	2026-06-19	true	Medium (3K-6K)
6	6	6	2023-06-26	Standard	3324	Active	2028-06-26	true	Medium (3K-6K)
7	7	7	2019-04-17	Standard	3300	Active	2026-07-03	true	Medium (3K-6K)
8	8	8	2022-05-06	Standard	2082	Active	2027-05-06	true	Low (≤3K)
9	9	9	2019-08-16	Standard	3747	Active	2026-07-03	true	Medium (3K-6K)
10	10	10	2019-01-28	Standard	3734	Active	2026-07-03	true	Medium (3K-6K)

5. Cross-Tab Check

Now let’s cross-tab *card_tier* and *credit_limit_bin* to check for data quality issues.



	card_tier text	credit_limit_bin text	count bigint
1	Black	Very High (20K+)	47
2	Gold	Medium (3K-6K)	47
3	Gold	Upper-Mid (6K-10K)	212
4	Platinum	High (10K-20K)	93
5	Standard	Low (≤3K)	171
6	Standard	Medium (3K-6K)	163

Standard cards have low to medium limits; Gold ones have medium to upper-mid limits; Platinum cards have high limits; and Black cards have very high limits. This result is expected.

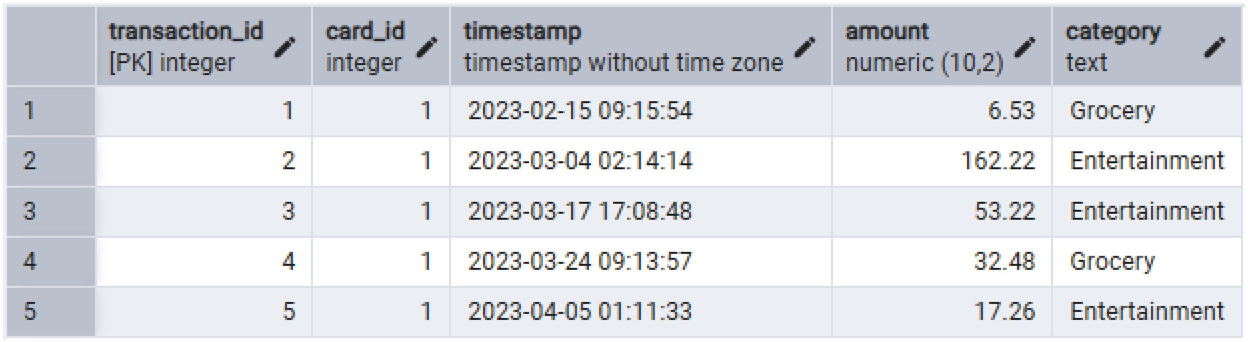
Also, the frequencies look correct: the lower the tier, the higher the number of cards. The Gold tier has fewer medium limits than upper-mids, demonstrating that this tier favours the upper-mid limits, leaving the medium ones for the Standard card. Similarly, Platinum cards cover high limits, while Black cards are oriented to very-high ones.

6. Conclusion

- No duplicates, missing values, unusual values, or consistency problems were identified in the dataset columns.
- A view was created, adding *credit_limit_bin* for further EDA and visualization.

Table: transactions

1. Overview

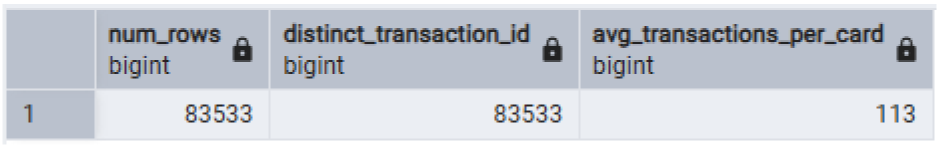


	transaction_id [PK] integer	card_id integer	timestamp timestamp without time zone	amount numeric (10,2)	category text
1	1	1	2023-02-15 09:15:54	6.53	Grocery
2	2	1	2023-03-04 02:14:14	162.22	Entertainment
3	3	1	2023-03-17 17:08:48	53.22	Entertainment
4	4	1	2023-03-24 09:13:57	32.48	Grocery
5	5	1	2023-04-05 01:11:33	17.26	Entertainment

The table records details about transactions made with each credit card, such as the amount and date.

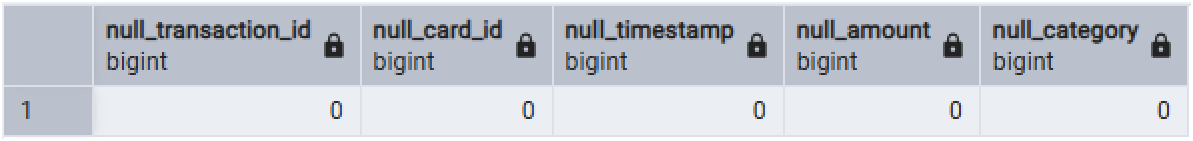
The table has the correct primary key (*transaction_id*) and foreign key (*card_id*) and contains 5 columns. The data types are correct. Timestamp is stored as TIMESTAMP, not just DATE, to reflect realistic transaction timing within each day.

2. Duplicates and Missing Values



	num_rows bigint	distinct_transaction_id bigint	avg_transactions_per_card bigint
1	83533	83533	113

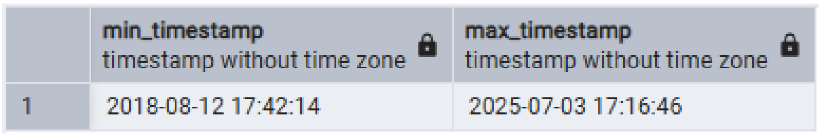
There are 83533 rows and the same number of distinct transaction IDs, validating the Primary Key. It corresponds to an average of 113 transactions per card. There are no duplicates, given that the number of rows and distinct IDs are the same.



	null_transaction_id bigint	null_card_id bigint	null_timestamp bigint	null_amount bigint	null_category bigint
1	0	0	0	0	0

There are no missing values (or NULLs).

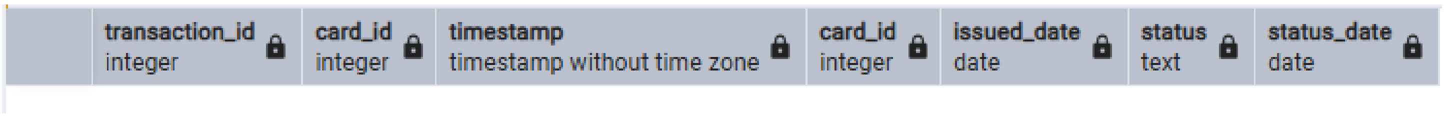
3. Column-by-Column Review



	min_timestamp timestamp without time zone	max_timestamp timestamp without time zone
1	2018-08-12 17:42:14	2025-07-03 17:16:46

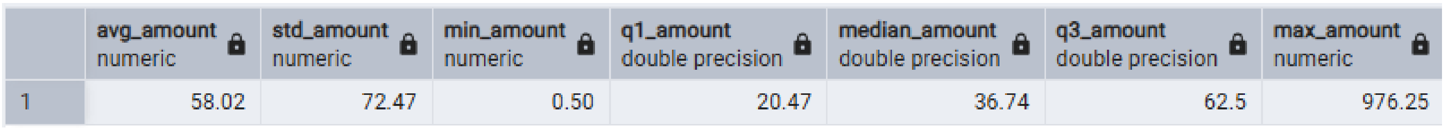
There are only recent transactions, from August 12, 2018, to July 3, 2025.

```sql
-- Checking for consistency in timestamp
SELECT
  t.transaction_id,
  t.card_id,
  t.timestamp,
  cc.card_id,
  cc.issued_date,
  cc.status,
  cc.status_date
FROM transactions t
JOIN v_credit_cards cc
ON t.card_id = cc.card_id
WHERE DATE(t.timestamp) < cc.issued_date  -- transaction before card issuance: not logical
OR DATE(t.timestamp) > CURRENT_DATE -- transaction in the future
OR (cc.status != 'Active' AND DATE(t.timestamp) > cc.status_date) -- transaction when the card was inactive
;
```



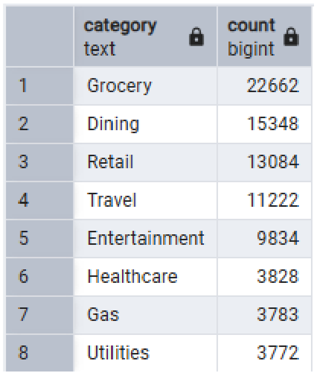
	transaction_id integer	card_id integer	timestamp timestamp without time zone	card_id integer	issued_date date	status text	status_date date

This script checks if there are transactions happening before card issuance, in the future, or after the day the card became inactive. The query is empty; therefore, all rules are obeyed.



	avg_amount numeric	std_amount numeric	min_amount numeric	q1_amount double precision	median_amount double precision	q3_amount double precision	max_amount numeric
1	58.02	72.47	0.50	20.47	36.74	62.5	976.25

The transactions have an average of $58.02 and can reach up to $976.25. This distribution is right-skewed because 75% of the transactions are below $62.50. The high standard deviation and maximum values confirm the distribution shape and variability in the final 25%.



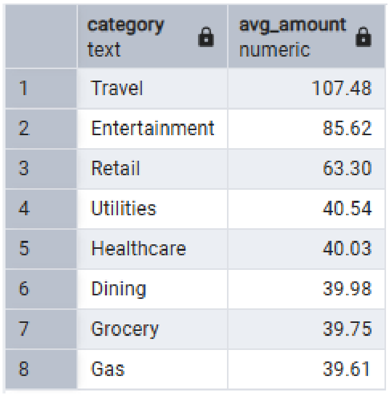
	category text	count bigint
1	Grocery	22662
2	Dining	15348
3	Retail	13084
4	Travel	11222
5	Entertainment	9834
6	Healthcare	3828
7	Gas	3783
8	Utilities	3772

The transactions are grouped into 8 categories, with *Grocery* having the most of them, and *Utilities*, the least.

4. Transformation and Grouping

No transformations or grouping were required at this stage for the transactions table.

5. Cross-Tab Check



	category text	avg_amount numeric
1	Travel	107.48
2	Entertainment	85.62
3	Retail	63.30
4	Utilities	40.54
5	Healthcare	40.03
6	Dining	39.98
7	Grocery	39.75
8	Gas	39.61

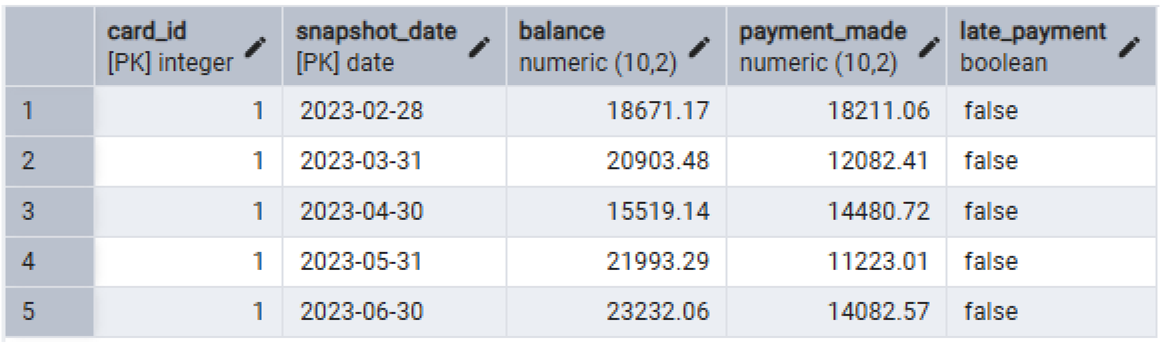
We can see that the customers spend more on Travel, Entertainment, and Retail, on average. They push balances up faster, raising utilization. Therefore, we can say these are risky categories, so we need to explore them better in the EDA.

6. Conclusion

- No duplicates, missing values, unusual values, or consistency problems were identified in the dataset columns.
- The categories Travel, Entertainment, and Retail are risky because customers tend to spend more on them, on average.

Table: payment_behavior

1. Overview



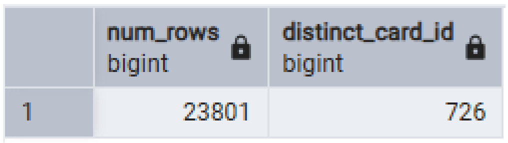
	card_id [PK] integer	snapshot_date [PK] date	balance numeric (10,2)	payment_made numeric (10,2)	late_payment boolean
1	1	2023-02-28	18671.17	18211.06	false
2	1	2023-03-31	20903.48	12082.41	false
3	1	2023-04-30	15519.14	14480.72	false
4	1	2023-05-31	21993.29	11223.01	false
5	1	2023-06-30	23232.06	14082.57	false

The table is a monthly snapshot of each credit card's status at the end of every billing cycle and whether the customer paid it late.

The table has the correct primary key (*card_id and snapshot_date*) and contains 5 columns. The data types are correct.

We can also note that *snapshot_date* corresponds to the last day of the month, when the billing cycle ends.

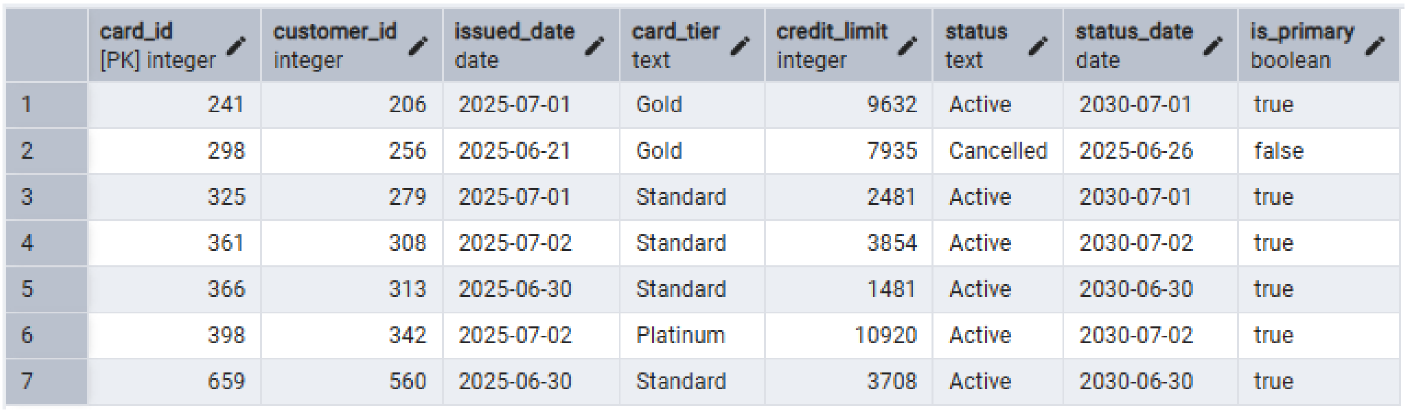
2. Duplicates and Missing Values



	num_rows bigint	distinct_card_id bigint
1	23801	726

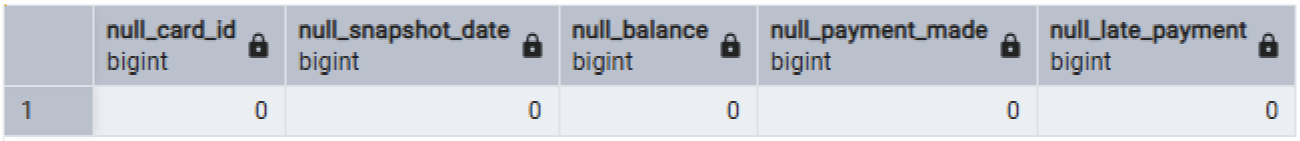
There are 23801 rows and 726 distinct card IDs, validating the Primary Key. However, the table credit_cards has 733 distinct card IDs. 7 cards are missing, so let's check them.

```
-- Checking the missing cards from payment_behavior
SELECT *
FROM credit_cards
WHERE card_id NOT IN (
  SELECT DISTINCT card_id FROM payment_behavior
);
```



	card_id [PK] integer	customer_id integer	issued_date date	card_tier text	credit_limit integer	status text	status_date date	is_primary boolean
1	241	206	2025-07-01	Gold	9632	Active	2030-07-01	true
2	298	256	2025-06-21	Gold	7935	Cancelled	2025-06-26	false
3	325	279	2025-07-01	Standard	2481	Active	2030-07-01	true
4	361	308	2025-07-02	Standard	3854	Active	2030-07-02	true
5	366	313	2025-06-30	Standard	1481	Active	2030-06-30	true
6	398	342	2025-07-02	Platinum	10920	Active	2030-07-02	true
7	659	560	2025-06-30	Standard	3708	Active	2030-06-30	true

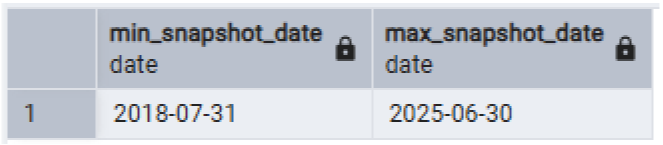
We saw in the *transactions* table that the most recent transaction happened on July 3, 2025. This is probably the date when the sample used in this analysis was created. The cards in all rows, except row 2, were recently issued and haven't completed one billing cycle yet. The card in row 2 was issued and cancelled 5 days later, and the reason goes beyond the scope of this project. Therefore, the 726 distinct card IDs are good for analysis.



	null_card_id bigint	null_snapshot_date bigint	null_balance bigint	null_payment_made bigint	null_late_payment bigint
1	0	0	0	0	0

There are no missing values (or NULLs).

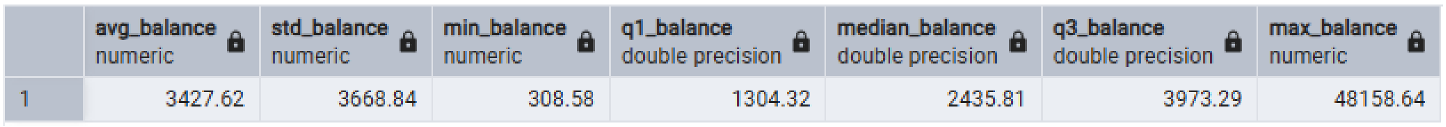
3. Column-by-Column Review



	min_snapshot_date date	max_snapshot_date date
1	2018-07-31	2025-06-30

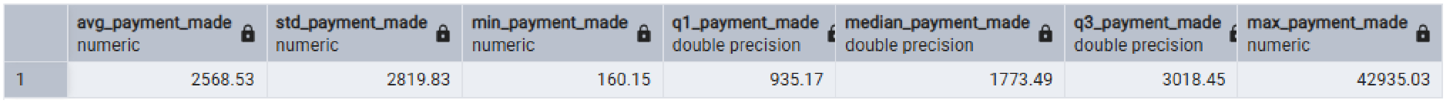
There are snapshots from July 31, 2018, to June 30, 2025.

The last snapshot date being at the end of June 2025, validates the conclusion we had about the 7 missing cards. All of them, except the cancelled one, were issued on or after this day. Therefore, they shouldn't be in this table as they haven't completed one billing cycle yet.



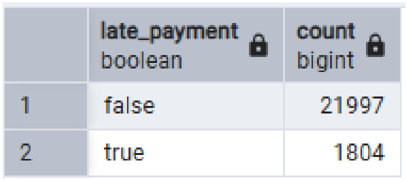
	avg_balance numeric	std_balance numeric	min_balance numeric	q1_balance double precision	median_balance double precision	q3_balance double precision	max_balance numeric
1	3427.62	3668.84	308.58	1304.32	2435.81	3973.29	48158.64

Balance represents the total amount the customer owes to the bank at the statement closing date. Its average is $3,427.62 and ranges from $308.58 to $48,158.64. This distribution is also right-skewed because 75% of the transactions are below $3,973.29. This is expected because we saw that most of the cards are Standard or Gold tiered, having lower limits. The high standard deviation and maximum values confirm the distribution shape and variability in the final 25%.



	avg_payment_made numeric	std_payment_made numeric	min_payment_made numeric	q1_payment_made double precision	median_payment_made double precision	q3_payment_made double precision	max_payment_made numeric
1	2568.53	2819.83	160.15	935.17	1773.49	3018.45	42935.03

The average of *payment_made* being $2,568.53, lower than the *balance*, indicates that the customers usually don't fully pay their balance. This distribution is right-skewed too (also expected), and there's a considerable amount of variability in the final 25%.



	late_payment boolean	count bigint
1	false	21997
2	true	1804

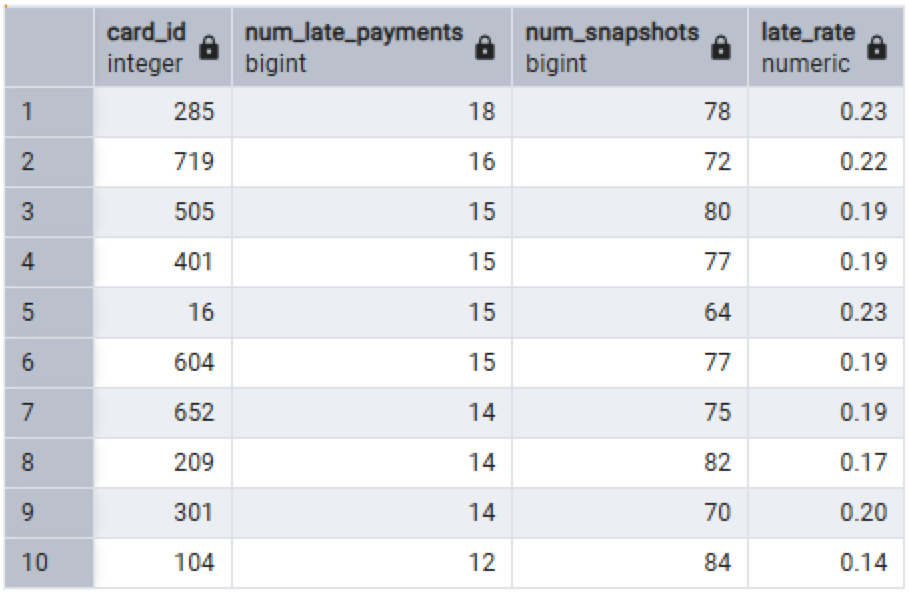
1804 statements were paid late, corresponding to 7.6% of all of them. This is the behaviour the bank wants to minimize.

4. Transformation and Grouping

No transformations or grouping were required at this stage for the payment_behavior table.

5. Cross-Tab Check

```sql
-- Checking for cross-tab consistency with late payment rate
SELECT
  pb.card_id,
  COUNT(*) AS num_late_payments,
  snapshots.num_snapshots,
  ROUND(COUNT(*) * 1.0 / snapshots.num_snapshots, 2) AS late_rate
FROM payment_behavior pb
JOIN (
  SELECT card_id, COUNT(*) AS num_snapshots
  FROM payment_behavior
  GROUP BY card_id
) AS snapshots ON pb.card_id = snapshots.card_id
WHERE pb.late_payment = TRUE
GROUP BY pb.card_id, snapshots.num_snapshots
ORDER BY num_late_payments DESC
LIMIT 10;
```



	card_id integer	num_late_payments bigint	num_snapshots bigint	late_rate numeric
1	285	18	78	0.23
2	719	16	72	0.22
3	505	15	80	0.19
4	401	15	77	0.19
5	16	15	64	0.23
6	604	15	77	0.19
7	652	14	75	0.19
8	209	14	82	0.17
9	301	14	70	0.20
10	104	12	84	0.14

This table shows the Top 10 card IDs with late payments and their *late_rate*, or the rate of the total months that the customer pays late. The owner of the card ID 285 paid 23% of the statements late.

These cross-tab results confirm that certain cards have consistently high late rates, which shows there is a real late payment behaviour pattern in the dataset. This aligns with the expected hidden signals, where risky spending and high utilization translate into repeated late payments for specific customer segments.

6. Conclusion

- No duplicates, missing values, unusual values, or consistency problems were identified in the dataset columns.
- 7 card IDs are not present in this table because they didn't complete one full billing cycle.
- The customers tend to pay the balance partially.
- 7.6% of the statements were paid late.
- Certain cards have consistently high late rates, showing a real late payment behaviour pattern in the dataset.

Data Validation & Profiling Conclusion

- The dataset passed all structural and logic checks.
- Relationships across customers, credit cards, transactions, and payment behaviour are consistent.
- Profiling confirms that certain segments show high spending and repeated late payments, which will be explored further in the EDA.

Next Steps

The next step is to go deeper into the data exploration phase. We will segment customers and cards, analyze utilization, late payment trends, and spending behaviour. The objective is to answer the business questions while still using SQL.