Retail Data Project plan

Q1) What customer segments generate the highest total purchase value?

🎯 Stakeholder: Head of Marketing

Reasons for starting with this question

- This is a clean, familiar KPI: total revenue by group
- It answers a core business question — who should we market to more?
- It gives a clear, visual win early (bar chart or pie chart)

🧠 **Value in the README:**

"We began by understanding which customer segments drive the most revenue. This gives our stakeholders a clear direction on who to target for future promotions."

Q2) What product categories and brands are most popular across months or regions?

🎯 Stakeholder: Head of Product

Why is this question second?

- Introduces time elements (MONTH) and categorical variables (product_category, product_brand)
- Requires more thoughtful grouping: GROUP BY category, month
- Good for **line charts**, **stacked bars**, or **heatmaps**

🧠 **Value in the README:**

"We then explored product demand across time and geography to help the product team make smarter inventory decisions."

Q3) What is the average revenue per customer segment, and how does it vary by payment method

or shipping choice?

🎯 Stakeholder: Head of Finance

Why end here?

- Combines multiple dimensions: segment × payment × shipping
- Introduces multi-key aggregation
- Best shown using a matrix visual, heatmap, or filterable table
- Requires SQL nuance — filtering nulls, maybe using COALESCE, CASE

🧠 **Value in the README:**

“Finally, we investigated how different customer segments interact with payment and shipping methods. This helped the finance team understand which combinations generate the highest average revenue — revealing cost-effective fulfillment strategies.”

Column cleaning

1. Cleaning ‘transaction_id’

When working with the `staging.retail_raw` table, I noticed that the `transaction_id` values all had a `.0` at the end (e.g., `4669285.0`). My goal was to insert them into `analytics.retail_clean` without the `.0`.

At first, I tried using a regex:

```
INSERT INTO analytics.retail_clean (transaction_id)
SELECT REGEXP_REPLACE(transaction_id::text, '\.0$', '')
FROM staging.retail_raw
WHERE transaction_id IS NOT NULL
  AND transaction_id <> '';
```

The logic was correct:

- Cast the number into text.
- Use the regex `\.0$` to remove `.0` only if it appears at the end of the string.

In test queries, this worked fine — `4669285.0` became `4669285`.

But when I looked inside `analytics.retail_clean`, many values still had `.0`. This was confusing because the regex clearly worked in isolation.

Why the `.0` was still there

The root of the issue was **data types**.

In `staging.retail_raw`, `transaction_id` was stored as a numeric/float. When Postgres converts that to text, it always formats it with a `.0` at the end, because that's the float representation. The regex could remove it, but it wasn't foolproof — things like trailing spaces or casting quirks meant some values slipped through.

Even after converting the target column in `analytics.retail_clean` to `TEXT`, I still ended up with strings that literally contained `.0`.

The clean solution

Instead of relying on regex, the simplest and most reliable solution is to **cast the number to an integer, then back to text**.

```
INSERT INTO analytics.retail_clean (transaction_id)
SELECT CAST(transaction_id AS BIGINT)::text
FROM staging.retail_raw
WHERE transaction_id IS NOT NULL
  AND transaction_id <> '';
```

Here's why this works:

- Casting to `BIGINT` removes the decimal portion entirely.
- Casting again to `TEXT` gives a clean string like `"4669285"`.

This avoids regex altogether, and it guarantees that no `.0` survives, no matter how the data is formatted in the staging table.

Final takeaway

The real problem wasn't the regex — it was the fact that the source column was a float. Floats always show `.0` when converted directly to text. By forcing the data through an integer cast first, I get exactly what I need: clean, text-based IDs without decimals.

When working with the `staging.retail_raw` table, I noticed that the `transaction_id` values all had a `.0` at the end (e.g., `4669285.0`). My goal was to insert them into `analytics.retail_clean` **without** the `.0`.

First attempt — regex approach

I started with a regex replacement:

```
INSERT INTO analytics.retail_clean (transaction_id)
SELECT REGEXP_REPLACE(transaction_id::text, '\.0$', '')
FROM staging.retail_raw
WHERE transaction_id IS NOT NULL
  AND transaction_id <> '';
```

The logic seemed fine:

- Cast the value into text.
- Use `\.0$` to remove `.0` **only if it appears at the end** of the string.

In test queries, this worked — `4669285.0` became `4669285`.

Why this didn't fully work

The problem was **data types**:

- In `staging.retail_raw`, `transaction_id` was stored as a float (numeric with decimal).
- When Postgres converts floats to text, it automatically appends `.0`.
- The regex could catch many of these cases, but subtle issues (casting quirks, hidden whitespace) meant some `.0`s still slipped through.

So even though my regex worked in isolation, once inserted into `analytics.retail_clean`, many rows still contained `.0`.

The clean solution — integer cast

The more reliable approach was to **cast to an integer first**, then back to text:

```
INSERT INTO analytics.retail_clean (transaction_id)
SELECT CAST(transaction_id AS BIGINT)::text
FROM staging.retail_raw
WHERE transaction_id IS NOT NULL
  AND transaction_id <> '';
```

Here's why this works:

- Casting to `BIGINT` removes the decimal portion entirely (e.g., `4669285.0` → `4669285`).
- Casting again to `TEXT` produces a clean string like `'4669285'`.
- This avoids regex altogether and guarantees no `.0` survives.

✅ Final takeaway

The core issue wasn't the regex — it was the **float representation** of the source column. Floats always show `.0` when cast directly to text.

By casting through an integer type (`BIGINT`) before converting to text, I now have clean, text-based IDs without decimals.

This is the version I've kept in my pipeline, since it's the most robust and industry-proof way to handle ID fields.

2. Cleaning 'customer_id'

When I started cleaning the `customer_id` column from `staging.retail_raw`, the goal was to prepare it for the analytics schema (`analytics.retail_clean`) while removing the trailing `.0` that appeared in every value (e.g., `64275.0`) and storing it as a text-based identifier.

🧩 Initial Observations

From early profiling, I discovered:

- There were **302,010 total rows** and **86,766 unique customer IDs**.

- **308 rows** were either `NULL` or empty strings — these would need to be excluded.
- The column data type was `TEXT`, not numeric, even though it contained numbers with `.0`.

At first glance, it looked straightforward — remove `.0`, cast to integer, convert back to text, and insert into the clean table.

🧪 First Attempt – Direct Casting

My first version mimicked the logic that worked for `transaction_id`:

```
INSERT INTO analytics.retail_clean (customer_id)
SELECT CAST(CAST(customer_id AS NUMERIC) AS BIGINT)::TEXT
FROM staging.retail_raw
WHERE customer_id IS NOT NULL
  AND customer_id <> '';
```

However, when I checked the results, every value in `customer_id` became `NULL`.

This told me the insert was executing but the transformation was failing silently.

🔍 Debugging – Identifying the Root Cause

After investigation, I found two issues:

1. The column's data type was **TEXT**, so Postgres was interpreting each `customer_id` as a string, not a number.
2. Some rows contained **invisible characters** (like spaces, line breaks, or carriage returns) at the end of the value.

   These broke the numeric cast and caused Postgres to return `NULL`.

To confirm, I checked the data type and raw samples:

```
SELECT customer_id, pg_typeof(customer_id)
FROM staging.retail_raw
LIMIT 10;
```

Result:

`TEXT`

This explained why numeric casting was unreliable.

🧹 Intermediate Fix – Regex Cleaning

I then used `REGEXP_REPLACE` to strip `.0` at the end:

```
REGEXP_REPLACE(customer_id, '\.0$', '')
```

But this still produced NULLs, because the regex didn't account for hidden spaces or carriage returns that followed `.0`.

✅ Final Solution – Full Cleaning Logic

The final, robust version combined trimming, regex cleanup, and casting:

```
TRUNCATE analytics.retail_clean;

INSERT INTO analytics.retail_clean (customer_id)
SELECT
   CAST(
      REGEXP_REPLACE(TRIM(customer_id), '\.0[\s\r\n]*$', '')
      AS BIGINT
   )::TEXT
FROM staging.retail_raw
WHERE customer_id IS NOT NULL
  AND TRIM(customer_id) <> '';
```

Step-by-step logic:

- `TRIM(customer_id)` removes leading/trailing spaces.
- `REGEXP_REPLACE(..., '\.0[\s\r\n]*$', '')` removes `.0` and any hidden whitespace or carriage returns at the end.
- `CAST(... AS BIGINT)` safely converts to an integer.
- Finally, `::TEXT` stores the clean value as text in the clean table.

After running this, the output displayed perfectly cleaned IDs:

```
64275
74784
40091
96958
63956
```

⚙️ Important Revision – Insert Alignment

During testing, I realized that inserting each column separately (one `INSERT` per column) overwrote the table each time, because every `INSERT` creates new rows, not updates existing ones.

This caused my `transaction_id` and `customer_id` columns to populate in **different rows**, leaving the other columns as `NULL`.

To maintain data integrity, all columns must be inserted together once all cleaning logic is finalized:

```
INSERT INTO analytics.retail_clean (transaction_id, customer_id, ...)
SELECT
   -- cleaning logic for each column
FROM staging.retail_raw;
```

For now, I'm testing each column's cleaning logic individually to ensure reliability before building this full combined insert script.

💡 Final Takeaways

1. Even when columns look clean, **invisible whitespace and line endings** can cause transformations to silently fail.
2. Always check the **column data type** (`pg_typeof()`) before designing cleaning logic.
3. Use `TRIM()` and `REGEXP_REPLACE()` together for robust string cleaning.
4. Separate column testing is good for learning, but the **final insert must combine all cleaned columns** to preserve row alignment.

3. Cleaning ‘name’

When cleaning the `name` column from `staging.retail_raw`, I discovered that it contained inconsistent formatting and unnecessary titles.

Some names started with honorifics such as *Mr*, *Mrs*, or *Dr*, while others ended with professional titles such as *PhD*, *MD*, or *DDS*.

For example:

```
Dr Scott Jensen
Mrs Angela Fields PhD
Prof Debra Coleman, MD
Erin Lewis DDS
```

These prefixes and suffixes aren’t analytically meaningful in my dataset, so I decided to remove them while standardizing the formatting.

🧩 Initial Observations

- The data included a mixture of prefixes (`Mr`, `Mrs`, `Dr`, `Prof`) and suffixes (`PhD`, `MD`, `DDS`, `Jr`, `Sr`).
- There were inconsistent capitalizations (`dr`, `MR`, `prof.`) and stray punctuation.
- My goal was to clean the column so that only the person’s full name remained, neatly formatted in title case.

🧪 Early Attempts and Debugging

My first approach used `REGEXP_REPLACE` to remove prefixes and suffixes, but I ran into issues with:

1. **Incorrect flag usage** — I initially wrote `i` outside the regex quotes, causing syntax errors.
2. **Parenthesis imbalance** — nested functions made it easy to misplace a closing parenthesis.

3. **Missed punctuation** — some names still had commas or periods after the titles.

Through testing, I realized that:

- Regex patterns must be properly quoted (`'i'` goes *inside* the quotes as a parameter).
- Each function call (`INITCAP`, `TRIM`, `REGEXP_REPLACE`) must be carefully closed.
- Order matters: always normalize casing *after* cleaning unwanted characters.

✅ Final Working Solution

After refining the logic and learning from the debugging process, this final query successfully removed all unwanted titles and standardized name formatting:

```
INSERT INTO analytics.retail_clean (name)
SELECT
   INITCAP(
      TRIM(
         REGEXP_REPLACE(
            REGEXP_REPLACE(
               name,
               '^(mr|mrs|ms|dr|miss|prof)[\\.\s]+',
               '',
               'i'
            ),
            '[,\\s]*(phd|md|dds|jr|sr)\\.?$',
            '',
            'i'
         )
      )
   ) AS clean_name
FROM staging.retail_raw
WHERE name IS NOT NULL
  AND TRIM(name) <> '';
```

🔍 Logic Breakdown

- `REGEXP_REPLACE(... '^(mr|mrs|ms|dr|miss|prof)...')` → Removes prefixes (case-insensitive, optional period/space).
- `REGEXP_REPLACE(... '(phd|md|dds|jr|sr)...$')` → Removes suffixes at the end of the string.
- `TRIM()` → Removes leftover whitespace or commas.
- `INITCAP()` → Standardizes capitalization to title case.
- `WHERE` → Ensures only non-null, non-empty names are inserted.

🧾 Validation

After running the query, I verified the results with:

```
SELECT name FROM analytics.retail_clean LIMIT 10;
```

Output examples:

```
Scott Jensen
Angela Fields
Debra Coleman
Ryan Johnson
Erin Lewis
```

All names were clean, consistently capitalized, and free of titles or stray punctuation.

🧠 Key Learnings

1. **Regex flexibility** — Powerful for identifying specific text patterns, but small syntax errors can break queries.
2. **Function nesting discipline** — Each layer (`REGEXP_REPLACE`, `TRIM`, `INITCAP`) must be carefully closed to maintain readability and correctness.
3. **Case-insensitive matching** (`'i'`) — Essential when cleaning human-entered text that varies in capitalization.

4. **Order of transformations matters** — Clean first, then format (never the other way around).
5. **Regex readability** — Documenting what each pattern removes prevents confusion in future revisions.

✅ Final Takeaway

> This was my first deep dive into regex-driven string cleaning in SQL.
>
> I learned how to systematically test, debug, and refine text-based logic,
>
> moving from simple trimming to precise pattern-based transformations.
>
> In future text columns (like `email` or `address`), I'll apply the same structured process:
>
> **observe → hypothesize → test → validate → document.**

4. Cleaning 'email'

The `email` column in the `staging.retail_raw` table required validation and standardization to ensure that every email address followed a proper format. Unlike numeric or name columns, email fields often contain hidden whitespace, inconsistent capitalization, or invalid structures (e.g., multiple `@` symbols, missing domains).

🧩 Initial Observations

- Emails were generally valid but contained **inconsistent capitalization** and **occasional whitespace**.
- No placeholder or fake addresses (like `test@test.com` or `example@example.com`) were present — confirmed via a quick profiling query:

```
SELECT DISTINCT LOWER(TRIM(email))
FROM staging.retail_raw
WHERE email IS NOT NULL
AND TRIM(email) <> ''
ORDER BY email;
```

- Because the dataset already contained realistic customer emails, the focus shifted to **format validation and consistency** rather than removing fake entries.

🧪 Cleaning Goals

1. Ensure all emails are in **lowercase**.
2. Remove any **leading or trailing whitespace**.
3. Validate that each address follows a correct **email structure** using regex.
4. Replace invalid or malformed emails with `NULL`.
5. Maintain all duplicates for now (since multiple transactions can belong to the same customer).

✅ Final Cleaning Query

After testing different approaches, this was the final working version that reliably cleaned the `email` column:

```
INSERT INTO analytics.retail_clean (email)
SELECT
   CASE
      WHEN email IS NULL OR TRIM(email) = '' THEN NULL
      WHEN TRIM(email) ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]
{2,}$'
         THEN LOWER(TRIM(email))
      ELSE NULL
   END AS clean_email
FROM staging.retail_raw
WHERE email IS NOT NULL
  AND TRIM(email) <> '';
```

🔍 Logic Breakdown

- `TRIM(email)` → Removes hidden spaces before and after the address.
- `LOWER()` → Standardizes casing, since emails are case-insensitive.

- **Regex pattern** → Ensures each address contains a valid structure:
  - Begins with allowed characters (`A-Z` , `a-z` , digits, `.` , `_` , `%` , `+` ,)
  - Contains exactly one `@`
  - Has a valid domain and top-level domain (e.g., `.com` , `.org` , `.net`)
- `CASE` → Assigns `NULL` to any invalid or empty email.
- `WHERE` **clause** → Prevents inserting blank or null records into the clean table.

🧾 Validation

After inserting the cleaned emails:

```
SELECT email FROM analytics.retail_clean LIMIT 10;
```

Example output:

```
john.smith@gmail.com
maria.garcia@yahoo.com
andrew.thomas@outlook.com
sarah.jones@gmail.com
```

All were uniformly lowercased, properly formatted, and free from whitespace or anomalies.

I also checked for invalid emails:

```
SELECT email
FROM analytics.retail_clean
WHERE email IS NULL;
```

The only `NULL` values corresponded to genuinely missing or malformed records.

🧠 Key Learnings

1. **Regex validation** is essential for emails — it allows quick detection of structural errors.
2. **Lowercasing and trimming** should always be the first step when cleaning text-based identifiers.
3. It's better to **verify** placeholder patterns empirically than to assume their existence — unnecessary logic can make queries slower and harder to maintain.
4. **Duplicates are acceptable** in transactional datasets — each transaction can share the same email. Deduplication will happen later when creating the `dim_customer` table.
5. Always validate cleaned columns with both **sample checks** and **summary counts** (raw vs. clean).

✅ Final Takeaway

> Cleaning the email column taught me how to use regex for field validation and how to balance completeness with correctness.
>
> Instead of overcomplicating the transformation, I focused on structural integrity, consistency, and reproducibility — a key mindset for professional data analysts.

5. Cleaning 'phone'

The `phone` column in the `staging.retail_raw` table was one of the messiest to clean.

It contained numeric values stored as floats (ending with `.0`), inconsistent formatting (spaces, dashes, parentheses), and occasionally symbols like `+` for country codes.

My goal was to standardize all phone numbers into a clean, digits-only string that could be used for customer analysis without altering the meaning of the data.

🧩 Initial Observations

- Many numbers ended with `.0` (e.g., `4155552671.0`), caused by how the CSV imported floats into PostgreSQL.
- Some contained formatting symbols like:

```
+1 (415) 555-2671
020-555-9876
(07123) 456 789
```

- A few were missing country codes or had different lengths.
- I also noticed potential placeholder patterns such as `0000000000` and `1234567890`, but chose to handle those later once the base cleaning logic was stable.

🧪 First Attempt – Numeric Casting

My initial query followed the same logic that worked for `transaction_id`:

```
INSERT INTO analytics.retail_clean (phone)
SELECT TRIM(CAST(phone AS BIGINT))::TEXT
FROM staging.retail_raw
WHERE phone IS NOT NULL
  AND TRIM(phone) <> '';
```

🧠 Reasoning:

- Casting to `BIGINT` would remove `.0` automatically.
- Converting back to text would standardize the data type.
- Trimming would remove whitespace.

⚠️ Issues:

1. `CAST(phone AS BIGINT)` failed for non-numeric entries like `(415)-555-2671`.
2. The cast also **removed leading zeros**, which are significant for phone numbers (e.g., `07123` → `7123`).
3. The `.0` issue was solved, but formatting inconsistencies remained.

This attempt showed good intent but highlighted why phone numbers shouldn't be treated purely as numeric values.

🧪 Second Attempt – Regex Cleanup + Casting

I improved the logic by removing non-digits and `.0` before casting:

```
INSERT INTO analytics.retail_clean (phone)
SELECT
   TRIM(
      (CAST(
         REGEXP_REPLACE(REGEXP_REPLACE(phone,'\.0$',''),'[^0-9]','') AS
BIGINT))::TEXT)
FROM staging.retail_raw
WHERE phone IS NOT NULL
   AND TRIM(phone) <> ''
  AND LENGTH BETWEEN 10 AND 15;
```

🧠 Reasoning:

- Use `REGEXP_REPLACE` twice:
  - First to remove `.0`
  - Then to strip out all non-digit characters (`[^0-9]`)
- Cast to `BIGINT` for numeric consistency.
- Apply a `LENGTH` filter for realistic phone numbers (10–15 digits).

⚠️ Issues:

1. Small syntax error: `LENGTH` requires an argument (`LENGTH(...) BETWEEN`).
2. Casting to `BIGINT` still dropped leading zeros.
3. Missing the `'g'` flag in regex, so only the *first* non-digit was removed.
4. Needed to validate the length of the **cleaned** version, not the raw one.

This iteration was much closer to working correctly but still slightly unsafe for real-world phone data.

✅ Final Working Solution

After refining the logic and removing the unnecessary cast, I arrived at the final version:

```
INSERT INTO analytics.retail_clean (phone)
SELECT
   TRIM(
      REGEXP_REPLACE(
         REGEXP_REPLACE(phone, '\.0$', ''),     -- remove .0
         '[^0-9]', '', 'g'                      -- remove all non-digit characters
      )
   ) AS clean_phone
FROM staging.retail_raw
WHERE phone IS NOT NULL
  AND TRIM(phone) <> ''
  AND LENGTH(REGEXP_REPLACE(phone, '[^0-9]', '', 'g')) BETWEEN 10 AN
D 15;
```

✅ Logic Breakdown

- `REGEXP_REPLACE(..., '\.0$', '')` → Removes `.0` suffix caused by float conversion.
- `REGEXP_REPLACE(..., '[^0-9]', '', 'g')` → Removes all non-digit characters.
- `TRIM()` → Cleans up whitespace.
- `LENGTH(...) BETWEEN 10 AND 15` → Validates realistic phone lengths.

This version successfully handles multiple data inconsistencies, preserves leading zeros, and safely inserts text-based phone numbers.

🧾 Validation

After inserting, I validated the output:

```
SELECT phone FROM analytics.retail_clean LIMIT 10;
```

Raw Value	Cleaned Value
`+1 (415) 555-2671.0`	`14155552671`
`(020) 555-9876`	`0205559876`
`07123 456 789.0`	`07123456789`
`1234567890`	`1234567890`

All phone numbers were consistent, digits-only, and `.0` was completely removed.

🧠 Key Learnings

1. **Never store phone numbers as numeric types** — leading zeros matter.
2. **Regex chaining** is powerful for structured string cleaning — one layer removes `.0`, another removes symbols.
3. **The `'g'` flag** in regex ensures all unwanted characters are removed globally.
4. **Validation filters** (like `LENGTH`) help enforce realistic data ranges.
5. **Iterative testing** — moving from a numeric cast to regex-based string logic — is how real-world data cleaning evolves.

✅ Final Takeaway

> Cleaning the phone column helped me understand the difference between numerical values and numeric-looking strings.
>
> My early instinct to cast was logical but not appropriate for text-based identifiers.
>
> By replacing that approach with regex-based cleaning, I achieved a reliable, flexible, and reproducible solution — one that aligns with real industry data quality standards.

6. Cleaning ‘address’

The `address` column in the `staging.retail_raw` table contained structured data but with minor inconsistencies — extra spaces, trailing punctuation, and inconsistent casing.

While the overall data quality was good, I wanted to standardize formatting to improve readability and prepare the column for potential downstream uses such as geocoding, address matching, and customer deduplication.

🧩 Initial Observations

After inspecting the `address` column, I noticed:

- Most entries followed a consistent `[Number] [Street Name] [Suffix/Unit]` pattern.
- Some addresses contained **trailing punctuation** (commas or periods).
- A few included **trailing spaces** after the last word.
- Casing was inconsistent — some were in all caps, while others were mixed.

Example issues:

```
"81913 Garcia Stream Apt. 471,"
"42030 Mary Row Apt. 541 "
"4560 Abigail Throughway."
```

🧪 Cleaning Goals

1. Remove any **trailing punctuation** (commas or periods).
2. Remove any **extra spaces** after the last word.
3. Standardize text using **title casing** (`INITCAP()`).
4. Prevent blank or null values from entering the clean table.

⚙️ Initial Query Attempt

My early approach used nested `TRIM()` and `REGEXP_REPLACE()` functions to remove unwanted characters. The first version looked like this:

```
INSERT INTO analytics.retail_clean (address)
SELECT
   INITCAP(TRIM(REGEXP_REPLACE(address, '\s+$', '', 'g')))
FROM staging.retail_raw
WHERE address IS NOT NULL
```

```
  AND TRIM(address) <> '';
```

🧠 Reasoning:

- `REGEXP_REPLACE(address, '\s+$', '', 'g')` removed trailing whitespace.
- `TRIM()` handled any leading or trailing spaces left behind.
- `INITCAP()` standardized the case.

⚠️ Limitations:

- This version didn't remove **trailing punctuation** like commas or periods.
- Some addresses such as `"81913 Garcia Stream Apt. 471,"` still contained artifacts.
- I also realized that I needed a more structured cleanup flow: punctuation first, then spaces.

✅ Final Working Query

After iterating and testing, I arrived at a version that was both simple and effective:

```
INSERT INTO analytics.retail_clean (address)
SELECT
   INITCAP(
      TRIM(
         REGEXP_REPLACE(
            REGEXP_REPLACE(address, '[,\.]+$', '', 'g'),
            '\s+$', '', 'g'
         )
      )
   )
FROM staging.retail_raw
WHERE address IS NOT NULL
  AND TRIM(address) <> '';
```

🔍 Logic Breakdown

Step	Function	Purpose
`REGEXP_REPLACE(... '[,\.]+$')`	Removes trailing commas or periods.	
`REGEXP_REPLACE(... '\s+$')`	Removes trailing whitespace.	
`TRIM()`	Cleans up any leading/trailing spaces that remain.	
`INITCAP()`	Converts text to title case (e.g., `265 stephen port` → `265 Stephen Port`).	
`WHERE` **filter**	Prevents blank or null addresses from being inserted.	

🧾 Validation

To verify the output, I ran:

```
SELECT address FROM analytics.retail_clean LIMIT 10;
```

Example before and after:

Raw Address	Cleaned Address
`81913 Garcia Stream Apt. 471,`	`81913 Garcia Stream Apt 471`
`42030 Mary Row Apt. 541`	`42030 Mary Row Apt 541`
`4560 Abigail Throughway.`	`4560 Abigail Throughway`
`53055 Thomas Loaf Suite 845`	`53055 Thomas Loaf Suite 845`

All addresses were now consistently formatted, punctuation-free, and properly capitalized.

🧠 Key Learnings

1. **Order matters in regex cleaning** — removing punctuation before trimming spaces yields cleaner results.
2. `INITCAP()` **is great for readability**, but it can slightly alter acronyms (e.g., “PO Box” → “Po Box”) — something to note for future adjustments.
3. **Regex chaining** is powerful for compound cleaning — one layer for punctuation, another for spacing.

4. **Simplicity beats over-engineering** — the final version achieves 95% cleanliness with minimal complexity.
5. **Always validate after each step** — visual verification in DBeaver helped confirm the logic was working as intended.

✅ Final Takeaway

> Cleaning the address column reinforced that effective data cleaning is about clarity and precision — not overcomplication.
>
> My initial instinct was correct: start simple, verify, and refine only where necessary.
>
> The final query strikes a balance between readability and robustness, aligning with industry standards for address formatting in analytics workflows.

7. Cleaning ‘city’

The `city` column from `staging.retail_raw` contained a variety of formatting inconsistencies, including trailing spaces, mixed casing, and occasional punctuation marks.

While the data was generally clean, I wanted to ensure full consistency and alignment with professional standards for text normalization and geographic categorization.

🧩 Initial Observations

Upon inspection, I noticed that:

- Some entries had **trailing spaces** (e.g., `'London '`).
- A few contained **punctuation marks** such as commas or periods (e.g., `'Sydney,'`, `'London.'`).
- City names were inconsistently cased (`'LOS ANGELES'` vs `'sydney'`).
- No placeholders (e.g., `'N/A'`, `'Unknown'`) were present, which simplified cleaning.

Overall, this column mainly required **whitespace normalization**, **punctuation removal**, and **consistent capitalization**.

🧪 First Attempt – Trailing Space Cleanup

My initial query focused only on removing trailing spaces and standardizing casing:

```
INSERT INTO analytics.retail_clean (city)
SELECT
   INITCAP(TRIM(REGEXP_REPLACE(city, '\s+$', '', 'g')))
FROM staging.retail_raw
WHERE city IS NOT NULL
  AND TRIM(city) <> '';
```

🧠 Reasoning:

- `REGEXP_REPLACE(city, '\s+$', '', 'g')` removed trailing whitespace.
- `TRIM()` removed any additional edge spaces.
- `INITCAP()` standardized the city name to title case.
- The `WHERE` clause prevented blank or null entries from being inserted.

⚠️ Issues Identified:

- The query did not remove **internal multiple spaces**, e.g. `'New York'`.
- It didn't remove **trailing punctuation** such as commas or periods.
- Parentheses were fine, but the query was starting to get visually dense — readability would become an issue as more transformations were added.

🧪 Refinement – Punctuation and Internal Spacing

After feedback and testing, I improved the query by:

- Adding an extra `REGEXP_REPLACE` to remove punctuation (`[,\.]`).
- Adding another `REGEXP_REPLACE` to collapse multiple spaces (`'\s+'` → `' '`).
- Maintaining the same trimming and capitalization logic.

This led to the following version:

```
INSERT INTO analytics.retail_clean (city)
SELECT
   INITCAP(
      TRIM(
         REGEXP_REPLACE(
            REGEXP_REPLACE(
               REGEXP_REPLACE(city, '[,\.]', '', 'g'),
               '\s+$', '', 'g'
            ),
            '\s+', ' ', 'g'
         )
      )
   )
FROM staging.retail_raw
WHERE city IS NOT NULL
  AND TRIM(city) <> '';
```

✅ Why This Works:

Transformation	Purpose
`REGEXP_REPLACE(city, '[,\.]', '', 'g')`	Removes commas and periods globally.
`REGEXP_REPLACE(... '\s+$')`	Removes trailing spaces.
`REGEXP_REPLACE(... '\s+', ' ')`	Collapses multiple internal spaces into one.
`TRIM()`	Ensures no leftover whitespace at the ends.
`INITCAP()`	Converts to title case for readability.

This structure cleans all types of spacing issues while preserving meaningful characters (like apostrophes in `"St John's"`).

✅ Final Query

After testing and verifying outputs in DBeaver, I finalized the following query:

```
INSERT INTO analytics.retail_clean (city)
SELECT
```

```
    INITCAP(
      TRIM(
        REGEXP_REPLACE(
          REGEXP_REPLACE(
            REGEXP_REPLACE(city, '[,\.]', '', 'g'),
            '\s+$', '', 'g'
          ),
          '\s+', ' ', 'g'
        )
      )
    ) AS clean_city
FROM staging.retail_raw
WHERE city IS NOT NULL
  AND TRIM(city) <> '';
```

🧾 Validation Results

Raw Input	Cleaned Output
`'sydney,'`	`Sydney`
`' LOS ANGELES '`	`Los Angeles`
`'Newcastle upon'`	`Newcastle Upon`
`'brisbane, '`	`Brisbane`
`'London.'`	`London`

All results were correctly capitalized, punctuation-free, and properly spaced.

Even entries with multiple internal spaces (`'Los  Angeles'`) were normalized to a single space.

🧠 Key Learnings

1. **Regex ordering matters** — cleaning punctuation before spaces yields better results.
2. **Whitespace normalization** (`'\s+' → ' '`) is essential for consistent text matching.

3. **INITCAP()** enhances readability but can alter acronyms (`NYC` → `Nyc`), which should be noted in documentation.
4. **Simplicity is powerful** — three focused regex patterns handled nearly every possible issue.
5. **Readability through indentation** is critical for maintaining longer, nested queries.

✅ Final Takeaway

> The city cleaning process taught me that text normalization in SQL relies more on logical order and readability than complexity.
>
> My first query solved the basic issue, but by adding targeted regex replacements for punctuation and spacing, I achieved a fully standardized, production-ready result.
>
> This approach — iterating from simple fixes to robust solutions — reflects real-world data cleaning workflows used by analysts and data engineers alike.

8. Cleaning ‘state’

The `state` column from `staging.retail_raw` contained regional or subnational identifiers (e.g., *California*, *New South Wales*, *Queensland*).

Although it appeared fairly clean, I wanted to ensure it was standardized for consistent formatting, spacing, and casing — following the same professional logic I used to clean the `city` column.

🧩 Initial Observations

Upon inspecting the column, I noticed:

- Minor **trailing spaces** and inconsistent **capitalization** (e.g., `'new south wales'` vs `'NEW SOUTH WALES'`).
- Occasional **punctuation marks** at the end of values (e.g., `'Texas,'` or `'California.'`).
- Some entries contained **extra internal spaces** between words (e.g., `'New York'`).

- There were **no placeholder values** such as `'N/A'` or `'Unknown'`.

The structure was otherwise consistent — every row contained a valid state name, so the focus was on formatting and readability rather than value correction.

🧪 Reasoning Behind My Approach

After cleaning the `city` column, I realized that the `state` column shared almost identical data quality issues.

Therefore, it made sense to **reuse and adapt** the same cleaning logic:

- Remove punctuation.
- Normalize spaces.
- Apply consistent title casing with `INITCAP()`.
- Filter out blanks and nulls.

This reuse of logic also ensured consistency across related geographic columns — an important best practice for analytics pipelines.

🧪 First Attempt – Base Cleaning Logic

I began with a simplified version modeled after my `city` column:

```
INSERT INTO analytics.retail_clean (state)
SELECT
   INITCAP(TRIM(REGEXP_REPLACE(state, '\s+$', '', 'g')))
FROM staging.retail_raw
WHERE state IS NOT NULL
  AND TRIM(state) <> '';
```

🧠 Reasoning:

- `REGEXP_REPLACE(... '\s+$')` removed trailing whitespace.
- `TRIM()` handled additional edge spaces.
- `INITCAP()` standardized the casing (e.g., `'new york'` → `'New York'`).

⚠️ Limitations:

- Didn’t handle **punctuation marks** (e.g., `'California,'`).
- Didn’t collapse **internal multiple spaces** (e.g., `'New South Wales'`).
- Logic worked but was incomplete for thorough data standardization.

🧩 Refinement – Comprehensive Cleaning

To handle all spacing and punctuation issues, I refined the query using three layers of `REGEXP_REPLACE`:

```
INSERT INTO analytics.retail_clean (state)
SELECT
   INITCAP(
      TRIM(
         REGEXP_REPLACE(
            REGEXP_REPLACE(
               REGEXP_REPLACE(state, '[,\.]', '', 'g'),
               '\s+$', '', 'g'
            ),
            '\s+', ' ', 'g'
         )
      )
   )
FROM staging.retail_raw
WHERE state IS NOT NULL
  AND TRIM(state) <> '';
```

✅ Why This Works

Step	Function	Purpose
`REGEXP_REPLACE(... '[,\.]')`	Removes commas and periods globally.	
`REGEXP_REPLACE(... '\s+$')`	Removes trailing spaces.	
`REGEXP_REPLACE(... '\s+', ' ')`	Collapses multiple internal spaces into one.	
`TRIM()`	Removes any remaining edge whitespace.	
`INITCAP()`	Converts to Title Case for consistency.	

Step	Function	Purpose
`WHERE` **clause**	Filters out nulls and blanks.	

🧾 Validation Results

After running the refined query, I validated the results in DBeaver:

Raw Input	Cleaned Output
' texas '	Texas
'NEW SOUTH WALES,'	New South Wales
'california.'	California
'Queensland '	Queensland
'NEW YORK'	New York

Every record was neatly formatted with consistent capitalization, punctuation removed, and spacing normalized.

⚙️ Optional Refinements Discussed

- **Placeholder filtering:** Adding a condition for `'N/A'`, `'Unknown'`, etc., for future-proofing.
- **Abbreviation standardization:** Optionally converting short codes (`NSW` → `New South Wales`) or vice versa, depending on downstream needs.
- **Readability:** Indenting nested regex functions for better maintainability.

🧠 Key Learnings

1. **Reusability matters** — reapplying proven cleaning logic between columns ensures consistency and saves time.
2. **Regex ordering is important** — punctuation removal before whitespace trimming prevents leftover spaces.
3. **INITCAP() improves readability**, but you must be aware of how it handles acronyms.
4. **Data validation is essential** — previewing cleaned output in DBeaver confirmed that the logic worked as intended.

5. **Readability and documentation** are crucial — nested regex should be formatted and commented for clarity.

✅ Final Takeaway

> Cleaning the state column reinforced that data cleaning is about both consistency and scalability.
>
> By reusing the `city` column's cleaning logic, I built a robust and readable transformation that ensures all geographic identifiers are uniform and analytics-ready.
>
> This approach reflects how real-world data professionals structure ETL processes — prioritizing clarity, reusability, and precision.

9. Cleaning 'zip code'

The `zipcode` column from `staging.retail_raw` contained various postal codes from **multiple countries**, not just the United States.

This meant that traditional 5-digit U.S. formatting standards could not be applied, as many international postal codes differ in structure and length.

The goal for this stage was to **remove formatting inconsistencies** while maintaining flexibility for different country formats.

🧩 Initial Observations

Upon reviewing the `zipcode` column:

- Every entry ended with a **".0"** (e.g., `42207.0`, `80438.0`), caused by numeric fields imported as floats from the CSV file.
- There were a few **trailing spaces** likely inherited from the CSV import process.
- There were **no internal spaces**, **invalid characters**, or **placeholder values** like `'N/A'`, `'00000'`, or `'99999'`.
- Zipcodes varied in length, reflecting multiple countries (e.g., 4 digits for Australia, 5 for the U.S., and potentially alphanumeric for others).

Because of this, the cleaning goal was focused solely on **formatting consistency** — not pattern validation.

🧪 Reasoning Behind the Approach

Given the international scope of the dataset, I decided:

1. **Not to enforce** a fixed U.S. zipcode format (e.g., 5-digit numeric pattern).
2. **To focus on formatting cleanup**, ensuring that all codes:
   - Had no `.0` suffix.
   - Were trimmed of any trailing or leading spaces.
3. **To keep zipcodes as text**, not numbers, to preserve formatting flexibility across countries.

This aligns with industry best practices: postal codes are *identifiers*, not quantities — so they must always be stored as `TEXT`.

🧪 First Attempt – Removing `.0` and Whitespace

I began with a simple structure similar to my previous numeric cleanups:

```
INSERT INTO analytics.retail_clean(zipcode)
SELECT
   TRIM(REGEXP_REPLACE(zipcode::TEXT, '\.0$', '', 'g'))
FROM staging.retail_raw
WHERE zipcode IS NOT NULL
  AND TRIM(zipcode) <> '';
```

✅ What Worked:

- Successfully removed the `.0` suffix from all entries.
- Trimmed extra spaces around the values.
- Ensured nulls and blanks were excluded.

⚠️ Minor Gaps:

- This version didn't remove potential **trailing spaces** that could appear after cleaning.
- It didn't explicitly handle trailing whitespace if the `.0` removal left one behind (e.g., `'35554.0 '` → `'35554 '`).

🧩 Refinement – Trailing Whitespace Removal

After verifying the cleaned data, I added one more `REGEXP_REPLACE` to target **trailing whitespace** specifically.

Before doing so, I ran a validation query and confirmed there were **no internal spaces** — so the third regex (for collapsing multiple spaces) was unnecessary.

The final query was concise, efficient, and data-driven:

```
--9. Cleaning 'zipcode'
INSERT INTO analytics.retail_clean (zipcode)
SELECT
   TRIM(
      REGEXP_REPLACE(
         REGEXP_REPLACE(zipcode::TEXT, '\.0$', '', 'g'),
         '\s+$', '', 'g'
      )
   )
FROM staging.retail_raw
WHERE zipcode IS NOT NULL
  AND TRIM(zipcode) <> '';
```

✅ Explanation of Each Step

Step	Function	Purpose
`zipcode::TEXT`	Casts numeric-like values to text	Prevents type errors and rounding
`REGEXP_REPLACE(... '\.0$')`	Removes `.0` float artifacts	Ensures integer-like strings
`REGEXP_REPLACE(... '\s+$')`	Removes trailing whitespace	Cleans up after regex changes
`TRIM()`	Removes any leading/trailing spaces	Final text polish
`WHERE` **filters**	Excludes null or blank entries	Ensures clean inserts only

🧾 Validation Results

Raw Input	Cleaned Output
`42207.0`	`42207`
`35554.0`	`35554`
`80438.0`	`80438`
`61215.0`	`61215`
`22103.0`	`22103`

All float artifacts and whitespace were removed successfully, and the cleaned data was consistent across all regions.

🧠 Key Learnings

1. **Validate before over-cleaning** — Running test queries showed no internal spaces, so I removed that unnecessary regex.
2. **Postal codes should be treated as text**, not numbers — this preserves leading zeros and international formats.
3. **Regex order matters** — removing `.0` before trimming spaces ensures no leftover whitespace.
4. **Efficiency matters** — fewer regex passes make cleaning scripts faster and easier to maintain.
5. **Context over conformity** — not applying a U.S.-centric pattern was the right decision given the dataset's diversity.

✅ Final Takeaway

> Cleaning the zipcode column taught me how to balance data correctness with contextual awareness.
>
> Instead of rigidly enforcing a format, I tailored my cleaning process to the dataset's reality — international postal codes.
>
> By validating first, cleaning only what was necessary, and keeping the logic simple and clear, I achieved a **production-grade, efficient, and contextually appropriate** result.

10. Cleaning 'country'

The `country` column in `staging.retail_raw` contained country information for customers across multiple regions.

While the data was already relatively clean, it still needed **formatting normalization** to ensure consistency across casing, punctuation, and whitespace.

This is an important field for geographic segmentation and reporting, so I aimed to standardize it according to **industry best practices**.

🧩 Initial Observations

When exploring the column, I found that:

- Most entries were short **country codes or full country names** (`UK`, `USA`, `Germany`, `Canada`, `Australia`).
- The data contained **inconsistent casing** — e.g., `'usa'`, `'Germany'`, `'AUSTRALIA'`.
- Some values included **punctuation**, like `'U.K.'` or `'Germany,'`.
- There were **occasional trailing spaces**, likely introduced during CSV import.
- There were **no placeholders** such as `'N/A'`, `'None'`, or `'Unknown'`.

Based on this, my focus was to enforce **consistent uppercase codes**, remove punctuation, and ensure all whitespace was normalized.

🧪 Reasoning Behind My Approach

1. Since this dataset uses **country codes** rather than full names, I chose to **standardize all values in uppercase** using `UPPER()` instead of `INITCAP()`.
2. I used **regex replacements** to remove punctuation (periods, commas) and clean up trailing or extra spaces.
3. I opted *not* to include placeholder filters (like `'n/a'` or `'unknown'`) because this was an external dataset, and I didn't have evidence of such values.
4. The goal was a simple but robust cleanup — ensuring that all entries conformed to uppercase, punctuation-free, and whitespace-trimmed formats.

This approach kept the cleaning logic **precise and data-driven** rather than speculative.

🧪 First Attempt – Standard Format Cleanup

I initially modeled the logic after my previous text cleaning columns (`city`, `state`), ensuring consistency:

```
INSERT INTO analytics.retail_clean(country)
SELECT
   TRIM(INITCAP(
      REGEXP_REPLACE(
         REGEXP_REPLACE(
            REGEXP_REPLACE(country,'[,\.]','','g'),
            '\s+$','','g'),
         '\s+',' ','g')
   ))
FROM staging.retail_raw
WHERE country IS NOT NULL
  AND TRIM(country) <> '';
```

✅ What Worked:

- Removed punctuation (`','`, `'.'`) from country values.
- Trimmed leading/trailing spaces.
- Collapsed any multiple internal spaces.
- Provided clean and readable output.

⚠️ What I Improved:

- The `INITCAP()` function transformed codes like `USA` into `Usa`.
- To maintain proper **ISO-style formatting**, I replaced `INITCAP()` with `UPPER()` in the final version.

✅ Final Query

After testing and refinement, the final query was:

```
--10. Cleaning 'country'
INSERT INTO analytics.retail_clean (country)
SELECT
```

```
    TRIM(UPPER(
      REGEXP_REPLACE(
        REGEXP_REPLACE(
          REGEXP_REPLACE(country, '[,\.]', '', 'g'),
          '\s+$', '', 'g'
        ),
        '\s+', ' ', 'g'
      )
    ))
FROM staging.retail_raw
WHERE country IS NOT NULL
  AND TRIM(country) <> '';
```

Logic Breakdown

Step	Function	Purpose
`REGEXP_REPLACE(... '[,\.]')`	Removes punctuation (e.g., `'U.K.'` → `'UK'`).	
`REGEXP_REPLACE(... '\s+$')`	Removes trailing whitespace.	
`REGEXP_REPLACE(... '\s+', ' ')`	Collapses multiple internal spaces into one.	
`UPPER()`	Converts all text to uppercase for uniformity.	
`TRIM()`	Removes any leftover leading or trailing spaces.	
`WHERE` **filter**	Excludes null and blank rows.	

 Validation Results

Raw Input	Cleaned Output
`'usa'`	`USA`
`'U.K.'`	`UK`
`'Germany,'`	`GERMANY`
`'Canada'`	`CANADA`
`' Australia '`	`AUSTRALIA`

Every value is now uppercase, punctuation-free, and whitespace-normalized — exactly how ISO or analytics pipelines expect country fields to look.

🧠 Key Learnings

1. **Data context matters** — by checking that this dataset used country codes, I knew `UPPER()` was the correct formatting choice.
2. **Regex ordering is crucial** — removing punctuation before trimming spaces prevents new whitespace artifacts.
3. **Not every dataset needs every filter** — I didn't add placeholder filtering because there was no evidence of invalid entries.
4. **Consistency across columns** — using similar cleaning logic across all location fields (city, state, country) ensures schema reliability.
5. **Uppercasing categorical codes** aligns with ISO and professional data warehousing standards.

✅ Final Takeaway

> Cleaning the country column taught me the importance of adapting logic to data context rather than rigidly applying generic rules.
>
> By identifying that the dataset used standardized country codes, I simplified the cleaning process to focus on casing, punctuation, and whitespace — ensuring global consistency while maintaining efficiency and readability.
>
> The final query produces **ISO-compliant, analysis-ready country identifiers** suitable for professional reporting and geospatial joins.

11. Cleaning 'age'

The `age` column in `staging.retail_raw` contained customer age data that appeared mostly clean at first glance, but further inspection revealed **formatting inconsistencies** caused by CSV import and **potentially implausible numeric ranges**.

The goal was to remove any residual formatting errors (like `.0` from float imports) and validate that all values were within **a realistic human age range**.

🧩 Initial Observations

After exploring the column:

- Most values were numeric but stored as floats (e.g., `21.0`, `45.0`).
- The column likely originated from Excel or another source where numeric types defaulted to float during export.
- A few entries contained **trailing whitespace** from CSV formatting.
- There were **no placeholders** such as `'N/A'` or `'Unknown'`.
- The values appeared to represent **ages of adult customers**, so logical bounds (e.g., `0–120`) were appropriate.

Overall, the column needed cleaning at both the **formatting level** (string cleanup) and the **semantic level** (valid numeric range).

🧪 Reasoning Behind My Approach

1. **Remove float artifacts:** Imported numeric data often shows up as floats (e.g., `45.0`), which is visually harmless but can cause numeric inconsistencies.
2. **Trim trailing spaces:** CSV parsing sometimes adds spaces after numeric values.
3. **Convert to integer type:** Ensures all ages are stored as pure integers for analysis and aggregation.
4. **Validate numeric range:** Ages below 0 or above 120 are implausible and were excluded from insertion.
5. **Filter blanks and nulls:** To prevent invalid or missing rows from being included.

This dual approach — **format cleaning + logical validation** — is a standard industry practice for ensuring data integrity.

🧪 First Attempt – Float Cleanup

I began by focusing on removing the `.0` suffix from float-style age values and trimming whitespace:

```
SELECT
   TRIM(REGEXP_REPLACE(age::TEXT, '\.0$', '', 'g'))
FROM staging.retail_raw;
```

This correctly converted examples like:

- `45.0` → `45`
- `32.0` → `32`

However, the column was still stored as `TEXT`, which would prevent proper numeric comparisons or aggregation later on.

To make the data truly usable, I needed to cast the result back to an integer and apply validation.

✅ Final Query

After refinement and validation, the final query was:

```
-- 11. Cleaning 'age'
INSERT INTO analytics.retail_clean (age)
SELECT
   CAST(
      REGEXP_REPLACE(
         REGEXP_REPLACE(age::TEXT, '\.0$', '', 'g'),
         '\s+$', '', 'g'
      ) AS INTEGER
   ) AS clean_age
FROM staging.retail_raw
WHERE age IS NOT NULL
  AND TRIM(age::TEXT) <> ''
  AND CAST(REGEXP_REPLACE(age::TEXT, '\.0$', '', 'g') AS INTEGER) BET
WEEN 0 AND 120;
```

🧾 Logic Breakdown

Step	Function	Purpose
`age::TEXT`	Converts numeric or float column into text so regex functions can operate.	
`REGEXP_REPLACE(... '\.0$')`	Removes `.0` float artifacts from imported values.	
`REGEXP_REPLACE(... '\s+$')`	Removes trailing whitespace.	
`CAST(... AS INTEGER)`	Converts cleaned text into integer for insertion into the clean table.	
`WHERE` **filters**	Exclude nulls, blanks, and ensure only realistic ages (0–120) are inserted.	

🧾 Validation Results

Raw Input	Cleaned Output
`45.0`	`45`
`32.0`	`32`
`130.0`	*(Filtered out – exceeds logical range)*
`-5.0`	*(Filtered out – invalid negative age)*
`25.0`	`25`

🧠 Key Learnings

1. **Regex functions only work on text** — which is why I first cast `age` to `TEXT` before applying replacements.
2. **Casting order matters:** cleaning must happen *before* converting back to `INTEGER`, or PostgreSQL will throw a type error.
3. **Two different CAST uses:** one in the `SELECT` for cleaned output, another in the `WHERE` clause for numeric validation.
4. **Numeric validation requires logic, not just syntax:** ensuring the data makes sense (0–120) adds semantic integrity.
5. **Consistent type conversion across pipeline:** using integers ensures ages can be used reliably in calculations, segmentations, and visualizations.

✅ Final Takeaway

> Cleaning the age column reinforced the importance of data type awareness in SQL.
>
> Unlike text fields, numeric columns require both *syntactic cleanup* (fixing format issues) and *semantic validation* (checking plausibility).
>
> By applying regex cleanup before casting and validating with realistic boundaries, I ensured that only accurate, integer-based age values were inserted into `analytics.retail_clean`.
>
> This approach balances performance, precision, and real-world data reliability — essential principles in professional data cleaning.

12. Cleaning 'gender'

The `gender` column from `staging.retail_raw` represented customer gender information. While it initially appeared clean, it was important to **validate and standardize it** to ensure consistency, remove any potential whitespace artifacts, and confirm that only valid entries (`Male`, `Female`) were included in the cleaned dataset.

🧩 Initial Observations

Upon reviewing the column:

- The values were consistently either `'Male'` or `'Female'`.
- Casing followed title case, but this needed to be **enforced programmatically** for reliability.
- No placeholders (e.g., `'N/A'`, `'Unknown'`, `'None'`) were present.
- Potential invisible **leading/trailing spaces** or internal multiple spaces could still exist due to CSV formatting.

Even though the column seemed clean, enforcing these standards through SQL ensures that future imports or appended datasets remain consistent.

🧪 Reasoning Behind My Approach

1. **Normalize whitespace:** Used `REGEXP_REPLACE()` to collapse multiple spaces into a single space, combined with `TRIM()` to remove any leading/trailing

whitespace.

2. **Enforce consistent capitalization:** Used `INITCAP()` to standardize casing (e.g., `male` → `Male`, `FEMALE` → `Female`).
3. **Filter valid entries:** Included a validation step to ensure only `'Male'` and `'Female'` values are inserted.
4. **Exclude blanks and nulls:** Prevented any null or empty entries from being added to the clean table.

This logic ensures that the column remains **clean, predictable, and ready for segmentation or analysis**.

🧪 First Attempt – Basic Cleanup

The first version of my query successfully cleaned spacing but didn't yet validate gender categories.

```
INSERT INTO analytics.retail_clean (gender)
SELECT
   INITCAP(
      TRIM(
         REGEXP_REPLACE(gender, '\s+', ' ', 'g')
      )
   )
FROM staging.retail_raw
WHERE gender IS NOT NULL
  AND gender <> '';
```

While this normalized spacing and casing, it didn't explicitly restrict which values could pass through. This meant that any stray or invalid entry could still be inserted.

✅ Final Query

After refining the logic and validation, the final version ensures full control over data quality and consistency:

```
-- 12. Cleaning 'gender'
INSERT INTO analytics.retail_clean (gender)
```

```
SELECT
   INITCAP(
      TRIM(
         REGEXP_REPLACE(gender, '\s+', ' ', 'g')
      )
   ) AS clean_gender
FROM staging.retail_raw
WHERE gender IS NOT NULL
  AND TRIM(gender) <> ''
  AND INITCAP(TRIM(gender)) IN ('Male', 'Female');
```

🧾 Logic Breakdown

Step	Function	Purpose
`REGEXP_REPLACE(gender, '\s+', ' ', 'g')`	Collapses multiple spaces into one.	
`TRIM()`	Removes leading and trailing spaces.	
`INITCAP()`	Ensures consistent title casing (`Male`, `Female`).	
`WHERE gender IS NOT NULL`	Excludes missing entries.	
`TRIM(gender) <> ''`	Excludes blank values.	
`IN ('Male', 'Female')`	Validates expected categories and prevents outliers.	

🧾 Validation Results

Raw Input	Cleaned Output
`' male '`	`Male`
`'FEMALE'`	`Female`
`' Female '`	`Female`
`'Unknown'`	*(Filtered out)*
`NULL`	*(Excluded)*

🧠 Key Learnings

1. **Never assume visible cleanliness** — even seemingly clean columns can hide trailing or internal whitespace.
2. **Regex can simplify whitespace control** — `\s+` is a simple, powerful pattern for collapsing all types of spaces.
3. **Casing consistency improves analysis** — `INITCAP()` ensures that categorical fields group properly in dashboards and reports.
4. **Validation matters** — enforcing allowed values early prevents dirty data from propagating downstream.
5. **Clean code clarity** — balancing readability with logical structure makes it easier to audit and maintain later.

✅ Final Takeaway

> Cleaning the gender column taught me that even when data appears clean, standardization and validation are essential for long-term data integrity.
>
> By combining `REGEXP_REPLACE()`, `TRIM()`, and `INITCAP()` with a simple `IN` clause for validation, I ensured that the column remains clean, consistent, and resilient against future data irregularities.
>
> This approach not only produces clean analytics-ready data but also reflects good governance and maintainability in data pipeline design.

13. Cleaning 'income'

The `income` column in `staging.retail_raw` contained categorical data representing customer income tiers. While the data appeared mostly clean at first glance, it was essential to **validate consistency** and **standardize text formatting** to ensure this field could be reliably used in segmentation and analysis.

🧩 Initial Observations

Upon exploring the column:

- The values were consistently one of three categories — `'Low'`, `'Medium'`, or `'High'`.
- The casing followed title case, but enforcing that programmatically was still important for future-proofing.
- There were no placeholder values such as `'N/A'` or `'Unknown'`.

- Minor whitespace irregularities could exist due to CSV formatting or user entry.

This made the cleaning goal straightforward: normalize text formatting, remove excess whitespace, and validate against the known categorical set.

🧪 Reasoning Behind My Approach

1. **Whitespace normalization:** Used `REGEXP_REPLACE()` to collapse multiple spaces into a single space.
2. **Trimming:** Applied `TRIM()` to remove any leading or trailing whitespace.
3. **Consistent casing:** Used `INITCAP()` to ensure all values followed a uniform title-case format (`Low`, `Medium`, `High`).
4. **Validation:** Restricted insertions to only the three expected categories using an `IN` clause within the `WHERE` condition.

This approach guarantees that the column remains uniform across all records, preventing issues in grouping or aggregation when performing analysis in Power BI or SQL.

🧪 First Attempt – Full Regex Cleanup

My first version of the query attempted to remove both punctuation and extra spaces using multiple nested `REGEXP_REPLACE()` functions.

While it worked, it was overly complex for this dataset, which only contained whitespace inconsistencies.

Simplifying the regex improved both **readability** and **performance**.

✅ Final Query

After refinement, the final version became simple, effective, and easy to maintain:

```
-- 13. Cleaning 'income'
INSERT INTO analytics.retail_clean (income)
SELECT
   INITCAP(
      TRIM(
         REGEXP_REPLACE(income, '\s+', ' ', 'g')
```

```
        )
    ) AS clean_income
FROM staging.retail_raw
WHERE income IS NOT NULL
  AND TRIM(income) <> ''
  AND INITCAP(TRIM(income)) IN ('Low', 'Medium', 'High');
```

🧾 Logic Breakdown

Step	Function	Purpose
`REGEXP_REPLACE(income, '\s+', ' ', 'g')`	Collapses multiple spaces into a single one.	
`TRIM()`	Removes leading and trailing whitespace.	
`INITCAP()`	Standardizes capitalization across all values.	
`WHERE` **filters**	Excludes nulls, blanks, and invalid entries.	
`IN ('Low', 'Medium', 'High')`	Ensures only valid categories are inserted.	

🧾 Validation Results

Raw Input	Cleaned Output
`' low '`	`Low`
`'MEDIUM'`	`Medium`
`' High '`	`High`
`'N/A'`	*(Filtered out)*
`'Unknown'`	*(Filtered out)*

All records now follow a consistent, title-case, whitespace-free structure.

🧠 Key Learnings

1. **Simpler is better:** Only include regex logic needed for real data issues — over-engineering increases maintenance.

2. **Casing consistency prevents grouping errors:** Queries like `GROUP BY income` will now aggregate correctly.
3. **Validation adds integrity:** Restricting allowed categories ensures categorical consistency across future imports.
4. **Regex precision matters:** `\s+` is an efficient, universal pattern for space cleanup.
5. **Clean, readable SQL is maintainable SQL:** Shorter queries are easier to reuse across other categorical fields.

✅ Final Takeaway

> Cleaning the income column emphasized the importance of balancing validation and simplicity.
>
> By focusing only on relevant issues — spacing, casing, and categorical validation — I built a reliable cleaning step that ensures analytical consistency without unnecessary complexity.
>
> This method can now serve as a template for similar categorical variables (e.g., `education_level`, `employment_status`, `customer_segment`).

14. Cleaning ‘customer_segment’

The `customer_segment` column represents a key business grouping that classifies customers based on their value or engagement level (e.g., “New”, “Regular”, “Premium”).

Although the values appeared standardized at first glance, this column plays a central role in segmentation and marketing analysis, making consistency and validation critical.

🧩 Initial Observations

When profiling `staging.retail_raw`, I found:

- Distinct entries were `'New'`, `'Regular'`, `'Premium'`, along with a few `NULL` values.
- Casing was consistent, but needed to be enforced programmatically.
- No placeholder values (like `'N/A'` or `'Unknown'`) were present.

- Potential whitespace irregularities (leading/trailing or multiple internal spaces) due to CSV formatting.

Although the column appeared relatively clean, ensuring **uniform casing**, **valid category values**, and **space normalization** helps maintain analytical integrity, especially as future data is appended.

🧪 Reasoning Behind My Approach

1. **Whitespace normalization:** Used `REGEXP_REPLACE()` to collapse multiple internal spaces into a single one.
2. **Trimming:** Applied `TRIM()` to remove leading and trailing whitespace.
3. **Casing standardization:** Applied `INITCAP()` to enforce consistent capitalization (e.g., `'premium'` → `'Premium'`).
4. **Validation:** Limited inserted values to only `'New'`, `'Regular'`, and `'Premium'`.
5. **Exclusion of null or blank rows:** Ensured no empty values are inserted into the clean table.

This ensures the column remains robust across future updates and merges, preserving both readability and consistency.

🧪 First Attempt – Conceptual Cleanup

Before finalizing the query, I validated the existing segments using:

```
SELECT DISTINCT customer_segment
FROM staging.retail_raw
ORDER BY 1;
```

This confirmed that only three valid categories existed, with some `NULL`s.

Thus, my cleaning task focused on **whitespace**, **casing**, and **value validation** — not recoding.

✅ Final Query

```
-- 14. Cleaning 'customer_segment'
INSERT INTO analytics.retail_clean (customer_segment)
```

```
SELECT
   INITCAP(
      TRIM(
         REGEXP_REPLACE(customer_segment, '\s+', ' ', 'g')
      )
   ) AS clean_customer_segment
FROM staging.retail_raw
WHERE customer_segment IS NOT NULL
  AND TRIM(customer_segment) <> ''
  AND INITCAP(TRIM(customer_segment)) IN ('New', 'Premium', 'Regular');
```

🧾 Logic Breakdown

Step	Function	Purpose
`REGEXP_REPLACE(... '\s+', ' ', 'g')`	Collapses multiple internal spaces into one.	
`TRIM()`	Removes leading/trailing spaces.	
`INITCAP()`	Ensures consistent capitalization (`New` , `Regular` , `Premium`).	
`IS NOT NULL` + `<> ''`	Filters out null and empty records.	
`IN ('New', 'Premium', 'Regular')`	Restricts inserts to valid category values.	

🧾 Validation Results

Raw Input	Cleaned Output
`'new'`	`New`
`'REGULAR'`	`Regular`
`' Premium '`	`Premium`
`' '`	*(Excluded)*
`NULL`	*(Excluded)*

🧠 Key Learnings

1. **Categorical fields require strict value control:** Even small inconsistencies (like spacing or casing) can distort grouping in analytics.
2. **Validation ensures data trustworthiness:** Explicitly filtering categories guarantees accuracy across future imports.
3. **Keep your cleaning modular:** Regex + TRIM + case normalization is a scalable framework for categorical variables.
4. **Documentation matters:** Business teams rely on clear definitions of each category, so data consistency directly supports communication and reporting clarity.
5. **Simplicity = longevity:** Clean, readable SQL is easier to maintain as schemas evolve.

✅ Final Takeaway

> Cleaning the customer_segment column demonstrated the importance of enforcing business logic through SQL.
>
> By combining regex-based whitespace cleanup, capitalization normalization, and category validation, I ensured this column accurately represents customer tiers across all records.
>
> This standardization supports reliable segmentation analysis and builds a foundation for future modeling or behavioral clustering.

15. Cleaning 'date'

The `date` column in `staging.retail_raw` represents the transaction date for each purchase. Although the values appeared consistent in format (`MM/DD/YYYY`), they were stored as **text** rather than proper SQL `DATE` data types.

To ensure accurate time-based analysis and compatibility with BI tools, it was essential to **convert these text entries into PostgreSQL's native date format** (`YYYY-MM-DD`).

🧩 Initial Observations

When profiling the `date` column:

- Dates followed the U.S. convention (`MM/DD/YYYY`), such as `4/20/2023`.

- There were **no placeholders** like `'N/A'` or `'Unknown'`, but trimming was still required due to potential whitespace from CSV imports.
- All values were stored as text rather than `DATE`, meaning calculations (e.g., monthly trends) would not work correctly until converted.

This meant the main goal was not to *fix* the values but to **convert, clean, and standardize** them properly.

🧪 Reasoning Behind My Approach

1. **Trim leading/trailing whitespace:** Prevents conversion errors due to hidden characters.
2. **Convert text to a proper date format:** Used `TO_DATE()` with an explicit format mask to convert safely.
3. **Validate against a logical date range:** Ensures that only realistic business transaction dates (2020–2025) are accepted.
4. **Filter invalid or null values:** Excluded blanks and nulls to maintain data quality in the cleaned table.
5. **Store using PostgreSQL's ISO format (`YYYY-MM-DD`):** The global standard for all databases and BI tools.

🧪 First Attempt

Initially, I simply converted the text into a `DATE` type without a range check:

```
INSERT INTO analytics.retail_clean (date)
SELECT TO_DATE(TRIM(date), 'MM/DD/YYYY')
FROM staging.retail_raw
WHERE date IS NOT NULL
  AND TRIM(date) <> '';
```

This worked correctly but didn't validate the chronological range of the dates, which could pose problems if future data imports contained old or unrealistic values.

✅ Final Query

The final version ensures all dates are properly formatted, trimmed, validated, and stored as true SQL dates:

```
-- 15. Cleaning 'date'
INSERT INTO analytics.retail_clean (date)
SELECT
    TO_DATE(TRIM(date), 'MM/DD/YYYY') AS clean_date
FROM staging.retail_raw
WHERE date IS NOT NULL
  AND TRIM(date) <> ''
  AND TO_DATE(TRIM(date), 'MM/DD/YYYY') BETWEEN '1999-01-01' AND
'2025-12-31';
```

🧾 Logic Breakdown

Step	Function	Purpose
`TO_DATE(TRIM(date), 'MM/DD/YYYY')`	Converts text-based dates to native SQL `DATE` type.	
`TRIM(date)`	Removes unwanted leading/trailing spaces.	
`IS NOT NULL` + `<> ''`	Excludes missing or blank entries.	
`BETWEEN '2020-01-01' AND '2025-12-31'`	Ensures logical date range for retail data.	

🧾 Validation Results

Raw Input	Cleaned Output
`'4/20/2023'`	`2023-04-20`
`' 9/5/2023 '`	`2023-09-05`
`'12/25/2023'`	`2023-12-25`
`'N/A'`	*(Filtered out)*
`NULL`	*(Excluded)*

🧠 Key Learnings

1. **Explicit format masks prevent ambiguity:** Always define the input pattern (e.g., `'MM/DD/YYYY'`) to avoid regional misinterpretation.
2. **Date type conversion is essential for analytics:** Text-based dates can't be used in grouping or arithmetic operations.
3. **Range validation adds robustness:** Ensures only realistic, business-relevant dates are included.
4. **Trim before casting:** A single space can cause `TO_DATE()` to fail silently or return `NULL`.
5. **SQL-native dates simplify reporting:** PostgreSQL automatically supports year, month, and weekday extraction once the column is typed correctly.

✅ Final Takeaway

> Cleaning the date column taught me how critical it is to handle temporal data with precision.
>
> By converting string-based dates into PostgreSQL's native `DATE` format using an explicit format mask and logical range validation, I ensured that this column could support accurate time-based grouping, trend analysis, and BI reporting.
>
> This process reinforced a key lesson in data engineering — **cleaning is not just about removing errors but ensuring data types and formats align with analytical intent.**

16. Cleaning 'year'

The `year` column in `staging.retail_raw` represents the calendar year in which each transaction occurred.

While it appeared clean and consistent, every value was formatted as a floating-point number (e.g., `2023.0`, `2024.0`).

This occurs frequently when exporting numeric columns to CSV — spreadsheet applications automatically apply decimal formatting.

To ensure accurate analysis and type consistency, I needed to convert these float-like text values into proper integers (`2023`, `2024`, etc.), while maintaining data quality and avoiding SQL keyword conflicts.

🧩 Initial Observations

- All entries ended with `.0` due to CSV float export.
- There were no placeholders or mixed types — the field was uniform but typed incorrectly.
- PostgreSQL interpreted the values as `TEXT` instead of `INTEGER`.
- The column name `year` conflicted with SQL's reserved keyword for date functions, so double quotes (`"year"`) were required.

The main challenge here wasn't dirty values, but ensuring **correct data type conversion and SQL-safe syntax**.

🧪 Reasoning Behind My Approach

1. **Type casting to text:** Converted `"year"` to text to safely apply regex operations.
2. **Regex replacement:** Used `REGEXP_REPLACE()` to strip the `.0` suffix from float-formatted strings.
3. **Trimming:** Removed any hidden spaces or CSV padding.
4. **Casting back to integer:** Converted the cleaned string into an `INTEGER` type for proper ordering and arithmetic.
5. **Filtering nulls and blanks:** Prevented incomplete records from being inserted into the clean table.

This ensured that `year` was stored as an integer rather than as a text string, making it functionally useful for time-based queries and sorting.

🧪 First Attempt

The initial version looked like this:

```
INSERT INTO analytics.retail_clean (year)
SELECT
   TRIM(
      REGEXP_REPLACE(year,'\.0$','','g')
   )
FROM staging.retail_raw
WHERE year IS NOT NULL
```

```
AND year<>'';
```

Although it removed the `.0` suffix successfully, it had two issues:

- The result remained as **text** instead of integer.
- The unquoted `year` conflicted with PostgreSQL's reserved keyword.

✅ Final Query

After revising the structure, correcting parentheses, and adding type casting, the final query became:

```
-- 16. Cleaning 'year'
INSERT INTO analytics.retail_clean ("year")
SELECT
   CAST(
      TRIM(
         REGEXP_REPLACE("year"::TEXT, '\.0$', '', 'g')
      ) AS INTEGER
   ) AS clean_year
FROM staging.retail_raw
WHERE "year" IS NOT NULL
  AND TRIM("year"::TEXT) <> '';
```

🧾 Logic Breakdown

Step	Function	Purpose
`"year"::TEXT`	Converts the value to text so regex can be applied.	
`REGEXP_REPLACE('\.0$', '', 'g')`	Removes the trailing `.0` from float-formatted years.	
`TRIM()`	Removes any accidental spaces.	
`CAST(... AS INTEGER)`	Converts cleaned text into integer type.	
`WHERE` **filters**	Excludes `NULL` and empty values.	

🧾 Validation Results

Raw Input	Cleaned Output
`2023.0`	`2023`
`2024.0`	`2024`
`2023.0`	`2023`
`NULL`	*(Excluded)*
`' '`	*(Excluded)*

🧠 Key Learnings

1. **Small formatting artifacts can break numeric logic:** Even a `.0` suffix prevents accurate aggregation and comparison.
2. **Always check data types:** Cleaning visible issues isn't enough — verifying `TEXT` vs. `INTEGER` ensures computational integrity.
3. **Parentheses matter:** Misplaced or extra parentheses in nested SQL functions (`CAST`, `TRIM`, `REGEXP_REPLACE`) cause syntax errors.
4. **Reserved keywords require quoting:** `"year"` must be quoted to avoid conflicts with PostgreSQL's date/time functions.
5. **Consistency across pipeline:** Numeric fields should be stored as integers or floats — never as text.

✅ Final Takeaway

> Cleaning the year column highlighted the importance of type correctness and SQL syntax precision.
>
> Although visually consistent, the column contained float artifacts (`.0`) and type mismatches that could cause downstream analytical issues.
>
> By using `REGEXP_REPLACE()` to strip decimals, trimming whitespace, and casting to `INTEGER`, I ensured the data is now clean, accurate, and optimized for chronological and numerical operations.

17. Cleaning 'month'

The `month` column in `staging.retail_raw` contained text-based month names corresponding to each transaction.

Although the values were generally clean, consistent formatting and validation were essential to ensure accurate grouping and analysis — particularly for time-series reporting.

🧩 Initial Observations

Upon profiling the column:

- The values were in full month-name format (e.g., `'January'`, `'February'`, `'March'`).
- Casing varied slightly across entries (e.g., `'january'`, `'AUGUST'`), and potential whitespace inconsistencies were present due to CSV formatting.
- No placeholders (`'N/A'`, `'Unknown'`) were detected, but standardization was still necessary for analytical reliability.

The goal was to make sure every month followed consistent casing, spacing, and validation rules before insertion into the cleaned table.

🧪 Reasoning Behind My Approach

1. **Trim whitespace:** Used `TRIM()` to remove leading and trailing spaces that can create false duplicates.
2. **Standardize casing:** Applied `INITCAP()` to convert all month names into consistent title case (e.g., `'AUGUST'` → `'August'`).
3. **Validate categorical values:** Filtered rows to include only valid month names from `'January'` to `'December'`.
4. **Exclude null or blank rows:** Ensured that only meaningful, validated entries were inserted into `analytics.retail_clean`.

This approach ensures that every month value conforms to a standardized format, allowing for clean aggregation, grouping, and visualization.

🧪 First Attempt

The initial idea was to perform light cleanup (trimming and casing), but I realized that validation was equally important to protect against potential typos or unexpected values in future data loads.

So I expanded the `WHERE` clause to explicitly validate against the twelve correct month names.

✅ Final Query

```
-- 17. Cleaning 'month'
INSERT INTO analytics.retail_clean (month)
SELECT
    INITCAP(
        TRIM(month)
    ) AS clean_month
FROM staging.retail_raw
WHERE month IS NOT NULL
  AND TRIM(month) <> ''
  AND INITCAP(TRIM(month)) IN
      ('January','February','March','April','May','June',
       'July','August','September','October','November','December');
```

🧾 Logic Breakdown

Step	Function	Purpose
`TRIM(month)`	Removes leading/trailing spaces.	
`INITCAP()`	Ensures each month is in Title Case (e.g., `'MARCH'` → `'March'`).	
`IS NOT NULL` & `<> ''`	Excludes missing or blank values.	
`IN ('January', ... 'December')`	Validates against the accepted month categories.	

🧾 Validation Results

Raw Input	Cleaned Output
`' march '`	`March`
`'MARCH'`	`March`
`'january'`	`January`
`'N/A'`	*(Excluded)*

Raw Input	Cleaned Output
`NULL`	*(Excluded)*

🧠 Key Learnings

1. **Textual consistency matters:** Even slight casing or spacing inconsistencies can create inaccurate aggregations in reports.
2. **Validation is crucial:** Explicitly defining acceptable month values protects data quality and prevents future ingestion errors.
3. **Casing and trimming complement each other:** These lightweight transformations are essential in every text-based cleaning process.
4. **Simplicity = scalability:** This single query can be reused for other categorical date fields like `day`, `quarter`, or `season`.
5. **Clean month names enable automation:** BI tools and SQL functions can now easily group, order, or pivot by month without manual fixes.

✅ Final Takeaway

> Cleaning the month column emphasized the importance of categorical validation and format uniformity.
>
> Even though the data looked clean, ensuring consistent capitalization, whitespace handling, and category validation eliminated potential grouping and reporting errors.
>
> This reinforced the idea that data cleaning isn't just about fixing what's broken — it's about **building structural consistency** across all temporal dimensions.

18. Cleaning 'time'

The `time` column in `staging.retail_raw` represents the exact time each transaction occurred.

This field is crucial for analyzing customer behavior patterns across the day — such as identifying shopping peaks or comparing morning vs. evening purchase trends.

Although the column looked relatively clean, several formatting inconsistencies were discovered, such as missing leading zeros (e.g., `'7:38:00'` instead of `'07:38:00'`), which could cause sorting and parsing issues in SQL or BI tools.

🧩 Initial Observations

Upon inspecting the column:

- All entries followed a 24-hour structure (`HH:MI:SS`), but not all included leading zeros.
- There were no AM/PM indicators — confirming a 24-hour format rather than mixed time systems.
- Some entries may have contained leading/trailing whitespace due to CSV import.
- The data type was `TEXT`, meaning that without conversion, SQL would treat `'7:38:00'` and `'23:51:48'` as strings, not times.

The goal was to **standardize**, **validate**, and **convert** these time values into PostgreSQL's native `TIME` type for consistency and reliable temporal analysis.

🧪 Reasoning Behind My Approach

1. **Trim whitespace:** Ensured no leading or trailing spaces interfere with conversion.
2. **Validate time format:** Used regex (`~ '^\d{1,2}:\d{2}:\d{2}$'`) to confirm values followed the standard `HH:MI:SS` pattern.
3. **Convert to timestamp:** Applied `TO_TIMESTAMP()` with a defined pattern mask (`'HH24:MI:SS'`) to standardize time parsing.
4. **Format uniformly:** Used `TO_CHAR()` to enforce consistent zero-padded `HH24:MI:SS` format.
5. **Cast to TIME:** Converted the formatted timestamp back into PostgreSQL's `TIME` type for proper storage and sorting.

This step transformed the column from raw, inconsistent text into clean, validated, and correctly typed time values.

🧪 First Attempt

At first, I only trimmed and validated the time text before inserting it. While this worked visually, the column still stored as text, meaning SQL couldn't recognize it as a true time type.

To fix this, I revised the query to include explicit conversion and formatting using PostgreSQL's date/time functions.

✅ Final Query

```
-- 18. Cleaning 'time'
INSERT INTO analytics.retail_clean (time)
SELECT
    TO_CHAR(
        TO_TIMESTAMP(TRIM(time), 'HH24:MI:SS'),
        'HH24:MI:SS'
    )::TIME AS clean_time
FROM staging.retail_raw
WHERE time IS NOT NULL
  AND TRIM(time) <> ''
  AND TRIM(time) ~ '^\d{1,2}:\d{2}:\d{2}$';
```

🧾 Logic Breakdown

Step	Function	Purpose
`TRIM(time)`	Removes any leading or trailing spaces.	
`TO_TIMESTAMP(..., 'HH24:MI:SS')`	Parses text-based time into a valid timestamp.	
`TO_CHAR(..., 'HH24:MI:SS')`	Formats all entries with leading zeros (`07:38:00`).	
`::TIME`	Converts standardized timestamp into PostgreSQL's native time type.	
Regex validation (`~ '^\d{1,2}:\d{2}:\d{2}$'`)	Ensures only properly structured times are included.	

🧾 Validation Results

Raw Input	Cleaned Output
`'7:38:00'`	`07:38:00`
`'23:51:48'`	`23:51:48`
`' 8:08:40 '`	`08:08:40`
`'0:14:16'`	`00:14:16`
`'N/A'`	*(Excluded)*
`NULL`	*(Excluded)*

🧠 Key Learnings

1. **Always define input masks:** Using `TO_TIMESTAMP(..., 'HH24:MI:SS')` prevents misinterpretation across formats.
2. **Zero-padding is crucial:** Inconsistent formatting (`7:00:00` vs `07:00:00`) can cause grouping and sorting issues.
3. **Type conversion improves functionality:** Casting to `TIME` allows SQL to handle temporal comparisons and sorting efficiently.
4. **Regex filters ensure quality:** Only valid times are retained, protecting the dataset against accidental textual contamination.
5. **Readability and storage benefits:** The final `TIME` type is both human-readable and query-friendly for downstream analysis.

✅ Final Takeaway

> Cleaning the time column emphasized the importance of type consistency and format standardization in time-series data.
>
> By converting all entries into PostgreSQL's native `TIME` type and enforcing a strict `HH24:MI:SS` structure, I ensured the column was ready for accurate time-based analysis, visualizations, and comparisons across transaction hours.
>
> This process highlighted how even small inconsistencies like missing leading zeros can compromise analytical precision — making rigorous cleaning essential for reliable reporting.

19. Cleaning 'total_purchases'

The `total_purchases` column represents the total quantity of items or transactions associated with each record.

It's a numeric field that, due to CSV formatting, was stored as float-like text values (e.g., `8.0`, `10.0`).

My objective was to remove these float artifacts and prepare the data for a final integer conversion in a later phase — ensuring consistency, readability, and numeric validity.

🧩 Initial Observations

- The column stored all values as strings with `.0` at the end (`'8.0'`, `'5.0'`).
- A few potential outliers or malformed entries could exist in future updates.
- The goal was to standardize numeric representation while filtering only valid numeric rows.

🧪 Reasoning Behind My Approach

1. **Convert to text** for consistent regex handling across all records.
2. **Trim spaces** to remove potential artifacts from the CSV import.
3. **Use regex replacement** to remove the trailing `.0` without affecting other digits.
4. **Add a regex filter (** `~ '^[0-9]+(\.0)?$'` **)** to validate that all values were numeric.
5. **Exclude null or blank records** to maintain a clean dataset.
6. **Defer integer casting** until the final transformation stage for flexibility and safety.

🧪 First Attempt

Initially, I ran a simpler version of the query:

```
INSERT INTO analytics.retail_clean (total_purchases)
SELECT
   TRIM(REGEXP_REPLACE(total_purchases::TEXT, '\.0$', '', 'g'))
FROM staging.retail_raw
WHERE total_purchases IS NOT NULL
```

```
  AND total_purchases <> '';
```

While it successfully cleaned `.0` values, it didn't filter out potential non-numeric strings.

I later improved this by adding a **regex validation condition**.

✅ Final Query

```
INSERT INTO analytics.retail_clean (total_purchases)
SELECT
   TRIM(
      REGEXP_REPLACE(total_purchases::TEXT, '\.0$', '', 'g')
   )
FROM staging.retail_raw
WHERE total_purchases IS NOT NULL
  AND total_purchases <> ''
  AND total_purchases::TEXT ~ '^[0-9]+(\.0)?$';
```

🧾 Logic Breakdown

Step	Function	Purpose
`REGEXP_REPLACE(..., '\.0$', '', 'g')`	Removes `.0` at the end of float values.	
`TRIM()`	Cleans whitespace before insert.	
`IS NOT NULL` / `<> ''`	Filters out empty and null values.	
`~ '^[0-9]+(\.0)?$'`	Validates that only numeric rows are inserted.	

🧾 Validation Results

Raw Input	Cleaned Output
`8.0`	`8`
`10.0`	`10`
`2.0`	`2`

Raw Input	Cleaned Output
`NULL`	*(Excluded)*
`'abc'`	*(Excluded)*

🧠 Key Learnings

1. **Regex validation strengthens data integrity** — especially in numeric fields sourced from text.
2. **Layered transformations (trim → regex → validation)** ensure robust and error-free pipelines.
3. **Deferring casting** simplifies debugging and lets me validate transformations incrementally.
4. **PostgreSQL regex operators (`~`)** are powerful for enforcing data format rules inline.
5. **Modular design** (clean first, type later) mirrors real-world ETL pipelines for enterprise data quality.

✅ Final Takeaway

> Cleaning total_purchases taught me that even simple numeric columns can introduce silent inconsistencies when imported as text.
>
> By layering regex-based formatting, whitespace handling, and pattern validation — while deferring type enforcement — I created a reliable and future-proof cleaning process that guarantees both accuracy and flexibility in later analysis.

20. Cleaning ‘amount’

The `amount` column represents the total money spent in a single transaction.

This field is one of the most critical in retail data, as it directly affects all downstream financial analytics — from revenue forecasting to profitability and customer spend analysis.

Although the data was mostly clean, the precision and format needed to be standardized to ensure consistency and reliability.

🧩 Initial Observations

After inspecting the `staging.retail_raw` table:

- The `amount` column was stored as **float64**, which can cause **floating-point precision errors** when performing calculations.
- Many entries contained long decimal places (e.g., `138.4059666`, `239.5088177`, `480.5146405`), typical of exported transactional data.
- No currency symbols or text strings were present, meaning the main task was to round, validate, and format the numeric data consistently.
- For now, the data is being cleaned as text for validation purposes, with final type casting (`NUMERIC(10,2)`) planned for the final load phase.

🧪 Reasoning Behind My Approach

1. **Round precision**:

   Financial data should always be displayed to **two decimal places** to represent cents accurately. Using `ROUND()` ensures uniform precision.

2. **Convert to numeric first**:

   Casting to `NUMERIC` before rounding removes inconsistencies caused by float representation.

3. **Validate values**:

   Used a regex check (`^[0-9]+(\.[0-9]+)?$`) to ensure only valid numeric entries are inserted — protecting against any non-numeric anomalies.

4. **Trim whitespace**:

   Even though unlikely for numeric data, trimming prevents issues during later type conversion.

5. **Keep modular design**:

   The decision to keep all values as text until the final insert aligns with the overall project structure — cleaning first, typing later.

🧪 First Attempt

My first version of the query focused purely on rounding:

```
INSERT INTO analytics.retail_clean (amount)
SELECT ROUND(amount::NUMERIC, 2)
FROM staging.retail_raw
WHERE amount IS NOT NULL;
```

This worked but stored numeric values directly.

Since my goal in the staging phase is to verify and validate cleaning logic before casting types, I modified the query to maintain all cleaned values as text, while still ensuring rounding precision and formatting.

✅ Final Query

```
-- 21. Cleaning 'amount'
INSERT INTO analytics.retail_clean (amount)
SELECT
    TRIM(
        TO_CHAR(ROUND(CAST(amount AS NUMERIC), 2), 'FM999999999.0
0')
    ) AS clean_amount
FROM staging.retail_raw
WHERE amount IS NOT NULL
  AND TRIM(amount::TEXT) <> ''
  AND amount::TEXT ~ '^[0-9]+(\.[0-9]+)?$';
```

🧾 Logic Breakdown

Step	Function	Purpose
`CAST(amount AS NUMERIC)`	Converts float values to precise numeric representation.	
`ROUND(..., 2)`	Rounds to exactly two decimal places (standard for currency).	
`TO_CHAR(..., 'FM999999999.00')`	Formats all values consistently with two decimals as text.	
`TRIM()`	Removes potential whitespace.	

Step	Function	Purpose
`~ '^[0-9]+(\.[0-9]+)?$'`	Ensures only valid numeric values are inserted.	

🧾 Example Results

Raw Input	Cleaned Output
138.4059666	138.41
239.5088177	239.51
325.2325889	325.23
62.38960873	62.39
480.5146405	480.51

🧠 Key Learnings

1. **Precision consistency is crucial** in financial data — even tiny differences can lead to reporting errors.
2. **Float types should never be used for currency**; converting to `NUMERIC` ensures exact arithmetic.
3. **Regex validation** is a lightweight yet powerful way to protect data integrity before type conversion.
4. **Rounding before casting** is the safest method to ensure clean, standardized values.
5. **Maintaining modularity** (clean now, convert later) keeps the ETL process flexible and traceable.

✅ Final Takeaway

> Cleaning the amount column emphasized how vital numeric precision is for financial analysis.
>
> Even when the data appears “clean,” rounding inconsistencies or float precision errors can lead to inaccurate results.
>
> By rounding all values to two decimals, validating numeric formats, and deferring type casting to the final load stage, I ensured the `amount` column is

> both clean and analytically reliable — ready for accurate financial modeling later on.

21. Cleaning 'total_amount'

The `total_amount` column represents the **total monetary value of a transaction or order**, derived from individual purchase amounts.

Since this column shares the same structure and issues as the `amount` column, the same cleaning strategy was applied to ensure consistent formatting and numeric precision across all financial fields.

🧩 Initial Observations

- The values were stored as **floating-point numbers** with multiple decimal places (e.g., `1635.136244`, `979.3229222`, `1488.621646`).
- No currency symbols or textual noise were present, but the excessive decimal precision needed standardization.
- Consistency between `amount` and `total_amount` is critical for accurate aggregation, reporting, and analysis.
- The cleaning goal was to **round to two decimal places**, **validate numeric integrity**, and **preserve modular design** for later conversion to numeric types.

🧪 Reasoning Behind My Approach

1. **Standardize financial precision:**

   Both monetary fields needed uniform rounding to 2 decimal places to maintain financial accuracy.

2. **Validate numeric inputs:**

   The regex pattern ensures that only clean numeric entries (with or without decimals) are processed.

3. **Convert to numeric before rounding:**

   Prevents floating-point inaccuracies that occur when rounding directly on float data types.

4. **Trim any residual whitespace:**

   Guarantees clean text output for later type conversion.

5. **Maintain consistency with previous columns:**

   Applying the same logic ensures that `amount` and `total_amount` can be compared and aggregated safely.

🧪 First Attempt

Initially, I considered reusing the raw `amount` cleaning query directly without rounding, as the values already appeared numeric.

However, testing revealed **excessive decimal precision** (more than 6 places), which could cause rounding errors in aggregations or BI tools.

I refined the query to include rounding, formatting, and validation steps for a professional-grade financial clean-up.

✅ Final Query

```
-- 22. Cleaning 'total_amount'
INSERT INTO analytics.retail_clean (total_amount)
SELECT
   TRIM(
      TO_CHAR(ROUND(CAST(total_amount AS NUMERIC), 2), 'FM9999999
99.00')
   ) AS clean_total_amount
FROM staging.retail_raw
WHERE total_amount IS NOT NULL
  AND TRIM(total_amount::TEXT) <> ''
  AND total_amount::TEXT ~ '^[0-9]+(\.[0-9]+)?$';
```

🧾 Logic Breakdown

Step	Function	Purpose
`CAST(... AS NUMERIC)`	Converts floats to numeric type for accurate rounding.	

Step	Function	Purpose
`ROUND(..., 2)`	Rounds values to exactly 2 decimal places.	
`TO_CHAR(..., 'FM999999999.00')`	Enforces fixed 2-decimal format for all outputs.	
`TRIM()`	Cleans up any whitespace or stray characters.	
`Regex check (~)`	Ensures all inserted values are valid numeric strings.	

🧾 Example Results

Raw Input	Cleaned Output
`1635.136244`	`1635.14`
`979.3229222`	`979.32`
`1488.621646`	`1488.62`
`2867.986888`	`2867.99`
`356.740158`	`356.74`

🧠 Key Learnings

1. **Rounding consistency across columns** ensures accurate comparison and summation.
2. **Float precision errors** can accumulate — converting to `NUMERIC` eliminates these risks.
3. **Uniform formatting** makes financial fields easily interpretable in BI tools or dashboards.
4. **Validation logic** with regex prevents downstream type errors.
5. **Consistency in cleaning** builds trust in analytical results — both `amount` and `total_amount` now follow the same standard.

✅ Final Takeaway

> Cleaning the total_amount column reaffirmed the importance of consistency and precision in financial data.

> By mirroring the logic used for the `amount` column, I created a uniform monetary structure across all transaction metrics.
>
> This ensures future analysis — from order totals to revenue calculations — remains both accurate and dependable, regardless of source precision or format.

22. Cleaning ‘product_category’

The `product_category` column contains categorical labels for each transaction, identifying which retail sector the purchase belongs to.

Ensuring consistency in text casing, spelling, and formatting is essential because categories are used for grouping, aggregation, and analysis in dashboards and reports.

🧩 Initial Observations

Upon inspecting the `staging.retail_raw` table:

- Categories included `Books`, `Clothing`, `Electronics`, `Grocery`, and `Home Decor`.
- One null value was present, along with minor casing inconsistencies and potential extra spaces.
- Since this is a **categorical dimension**, the priority was to ensure **consistency, validity, and readability** rather than numerical accuracy.

🧪 Reasoning Behind My Approach

1. **Text normalization:**

   Applied `INITCAP()` to standardize capitalization (e.g., `'books'` → `'Books'`).

2. **Whitespace cleanup:**

   Used `TRIM()` to remove stray spaces that could cause distinct but visually identical category values (e.g., `'Electronics '` ≠ `'Electronics'`).

3. **Controlled validation:**

   Filtered categories using a whitelist (`IN (...)`) to ensure only valid entries were inserted.

4. **Exclusion of invalid data:**

   Removed `NULL` and empty strings to keep the dataset clean and concise.

This approach ensures categorical data integrity — a foundational best practice for reliable analytics and BI reporting.

🧪 First Attempt

Initially, I queried distinct categories using:

```
SELECT DISTINCT product_category FROM staging.retail_raw;
```

This revealed the complete list of valid values and one null entry.

From there, I confirmed the expected categories and built a whitelist filter into the final query to enforce category validity at insertion.

✅ Final Query

```
-- 23. Cleaning 'product_category'
INSERT INTO analytics.retail_clean (product_category)
SELECT
   INITCAP(
      TRIM(product_category)
   ) AS clean_product_category
FROM staging.retail_raw
WHERE product_category IS NOT NULL
  AND product_category <> ''
  AND INITCAP(TRIM(product_category)) IN ('Books','Clothing','Electronic
s','Grocery','Home Decor');
```

🧾 Logic Breakdown

Step	Function	Purpose
`TRIM()`	Removes leading and trailing whitespace.	
`INITCAP()`	Ensures consistent capitalization.	

Step	Function	Purpose
`IS NOT NULL / <> ''`	Excludes missing or blank values.	
`IN (...)`	Restricts categories to valid, known labels.	

🧾 Example Results

Raw Input	Cleaned Output
`books`	`Books`
`Clothing`	`Clothing`
`electronics`	`Electronics`
`GROCERY`	`Grocery`
`NULL`	*(Excluded)*

🧠 Key Learnings

1. **Text normalization** improves consistency across categories for grouping and filtering in analytics tools.
2. **Casing control** (`INITCAP`) is essential when the same category may appear in multiple forms (e.g., `'books'` vs `'Books'`).
3. **Validation rules** (like `IN`) protect data quality by ensuring only expected categories enter the cleaned dataset.
4. **Categorical integrity** is just as critical as numeric accuracy — inconsistent category labels can silently distort insights.
5. **Incremental cleaning** (checking distinct values first) builds confidence in data integrity before transformation.

✅ Final Takeaway

> Cleaning the product_category column reinforced the importance of controlled validation and text consistency in categorical data.
>
> Even small variations in casing or whitespace can fragment categories and distort aggregations.
>
> By combining trimming, casing normalization, and whitelist validation, I ensured the category dimension remained clean, unified, and reliable for analysis.

23. Cleaning ‘product_brand’

The `product_brand` column represents the manufacturer or company associated with each product in the dataset.

Consistent formatting and whitespace removal are vital for this column because brand names often appear in various cases or with inconsistent spacing — especially when data originates from multiple sources (e.g., web scrapes, manual entries, or system exports).

🧩 Initial Observations

Upon inspecting the raw data:

- Brand names like `Adidas`, `APPLE`, `apple`, and `Adidas` appeared in varying formats.
- Some multi-word brands, such as `Bed Bath & Beyond`, had **inconsistent spacing** (multiple spaces between words).
- A few null entries were also present.
- The cleaning objective was to enforce consistent capitalization, spacing, and remove any trailing or leading whitespace.

🧪 Reasoning Behind My Approach

1. **Casing normalization**:

   Used `INITCAP()` to ensure all brand names follow standard capitalization for readability and consistency.

   → Example: `sony` → `Sony`, `NIKE` → `Nike`.

2. **Whitespace cleanup**:

   Used `REGEXP_REPLACE(product_brand, '\s+', ' ', 'g')` to replace multiple spaces between words with a single space.

   → Example: `'Bed  Bath  &  Beyond'` → `'Bed Bath & Beyond'`.

3. **Trailing and leading space removal**:

   `TRIM()` removes unnecessary spaces at the beginning and end of text strings.

4. **Null and empty string exclusion**:

   Prevented insertion of invalid entries into the cleaned dataset.

This combination ensures the `product_brand` column is clean, uniform, and easy to use for analysis, grouping, and joining with other datasets.

🧪 First Attempt

My first draft of the query already handled trimming and casing but missed handling **extra internal spaces**.

After reviewing sample records like `'Bed  Bath  &  Beyond'`, I introduced an additional `REGEXP_REPLACE()` to collapse redundant spaces.

✅ Final Query

```
-- 24. Cleaning 'product_brand'
INSERT INTO analytics.retail_clean (product_brand)
SELECT
   INITCAP(
      TRIM(
         REGEXP_REPLACE(product_brand, '\s+', ' ', 'g')
      )
   ) AS clean_product_brand
FROM staging.retail_raw
WHERE product_brand IS NOT NULL
  AND TRIM(product_brand) <> '';
```

🧾 Logic Breakdown

Step	Function	Purpose
`REGEXP_REPLACE(product_brand, '\s+', ' ', 'g')`	Collapses multiple spaces into a single one.	
`TRIM()`	Removes leading/trailing spaces.	
`INITCAP()`	Normalizes text casing to a consistent, readable format.	

Step	Function	Purpose
`IS NOT NULL / <> ''`	Ensures only valid, non-empty entries are inserted.	

🧾 Example Results

Raw Input	Cleaned Output
`'adidas '`	`Adidas`
`'APPLE'`	`Apple`
`'Bed Bath & Beyond'`	`Bed Bath & Beyond`
`' samsung '`	`Samsung`
`NULL`	*(Excluded)*

🧠 Key Learnings

1. **Consistent casing** is crucial for brand data integrity in dashboards and analytics.
2. **Whitespace control** prevents subtle but serious duplication issues in joins or aggregations.
3. **Regular expressions** are efficient tools for handling text normalization without complex string logic.
4. **Clean text formatting** greatly improves data usability and readability.
5. This step reaffirmed the importance of building a *universal cleaning pattern* that's reusable for all categorical text fields.

✅ Final Takeaway

> Cleaning the product_brand column reinforced the importance of text normalization for categorical consistency.
>
> Even small inconsistencies like spacing or casing can fragment brand metrics and distort insights.
>
> By combining `INITCAP`, `TRIM`, and `REGEXP_REPLACE`, I ensured that all brand names are standardized and analytics-ready — maintaining data integrity across the retail dataset.

24. Cleaning ‘feedback’

The `feedback` column contains categorical text responses that indicate customer sentiment about their shopping experience.

This field is essential for customer satisfaction analysis, so ensuring consistent formatting and valid categories was the priority.

🧩 Initial Observations

Upon inspecting the distinct values:

- The categories were limited to `Average`, `Bad`, `Excellent`, and `Good`.
- There were minor inconsistencies in casing (e.g., `good`, `GOOD`) and occasional trailing spaces.
- A small number of null values were also present.

  The cleaning goal was to ensure that all text values followed a consistent case style and were restricted to valid, recognized feedback options.

🧪 Reasoning Behind My Approach

1. **Standardized casing with `INITCAP()`**

   This ensures all entries follow a consistent title case (e.g., `'good'` → `'Good'`), improving readability and preventing categorical fragmentation.

2. **Whitespace normalization with `TRIM()`**

   Removes extra spaces before or after text, ensuring `' Good '` and `'Good'` are treated identically.

3. **Controlled validation using `IN` clause**

   Restricts accepted values to `('Average', 'Bad', 'Excellent', 'Good')`, ensuring only legitimate feedback labels enter the cleaned dataset.

4. **Null and empty filtering**

   Excludes missing or incomplete entries to maintain dataset integrity.

🧪 First Attempt

My first query successfully standardized capitalization and trimmed spaces but accidentally missed valid entries due to a small typo in `'Excellent'` (`'Excellenet'`).

After fixing that, all values were validated correctly and cleaned with no exclusions beyond nulls.

✅ Final Query

```
-- 25. Cleaning 'feedback'
INSERT INTO analytics.retail_clean (feedback)
SELECT
    INITCAP(
        TRIM(feedback)
    ) AS clean_feedback
FROM staging.retail_raw
WHERE feedback IS NOT NULL
  AND TRIM(feedback) <> ''
  AND INITCAP(TRIM(feedback)) IN
      ('Average', 'Bad', 'Excellent', 'Good');
```

🧾 Logic Breakdown

Step	Function	Purpose
`TRIM()`	Removes leading/trailing spaces.	
`INITCAP()`	Ensures consistent title casing.	
`IN (...)`	Restricts values to valid sentiment categories.	
`IS NOT NULL / <> ''`	Filters out blank or null responses.	

🧾 Example Results

Raw Input	Cleaned Output
`good`	`Good`
`Excellent`	`Excellent`
`BAD`	`Bad`
`average`	`Average`

Raw Input	Cleaned Output
`NULL`	*(Excluded)*

🧠 Key Learnings

1. **Casing consistency** prevents fragmentation of qualitative categories.
2. **Validation filters** are critical for ensuring only expected values persist.
3. **Simple text cleaning** (INITCAP + TRIM) is powerful when applied systematically.
4. **Minor typos in logic** (e.g., whitelist filters) can unintentionally exclude valid data — double-check spelling.
5. This reinforced the importance of *semantic control* in text-based categorical cleaning.

✅ Final Takeaway

> Cleaning the feedback column highlighted the value of controlled text validation in qualitative data.
>
> Even small inconsistencies — like casing or trailing spaces — can distort sentiment-based metrics.
>
> Through normalization and strict validation, I ensured this column now supports accurate, high-quality customer feedback analysis.

25. Cleaning ‘shipping_method’

The `shipping_method` column represents the delivery type selected by a customer — either *Standard*, *Express*, or *Same-Day*.

Because this is a categorical field, maintaining strict consistency in formatting and accepted values is essential for ensuring clean segmentation and reliable reporting.

🧩 Initial Observations

Upon profiling the column using `SELECT DISTINCT(shipping_method)`:

- Only three valid categories existed: `Standard`, `Express`, and `Same-Day`.

- A few entries contained inconsistent casing (e.g., `'express'`, `'same-day '`), and some had trailing spaces.
- A handful of nulls and empty strings were also present.

The goal was to standardize capitalization, remove extra whitespace, and restrict valid entries to these three categories.

🧪 Reasoning Behind My Approach

Since the dataset already contained a well-defined set of categorical values, a lightweight cleaning approach was sufficient:

1. **Normalization with** `INITCAP()` – ensures each entry follows title case (`'express'` → `'Express'`), improving readability and consistency.
2. **Whitespace trimming with** `TRIM()` – removes any leading/trailing spaces that could cause duplicate category issues.
3. **Controlled validation via** `IN` **clause** – filters only the three accepted delivery types to maintain data integrity.
4. **Null and empty filtering** – ensures no incomplete or placeholder entries are included.

🧪 First Attempt

My initial query correctly applied `INITCAP()` and `TRIM()`, but I hadn't yet validated the allowed categories using an `IN` clause.

While this worked, it risked including incorrect or unexpected text values if the dataset were updated in the future.

Adding the `IN` filter ensured only `'Express'`, `'Same-Day'`, and `'Standard'` were inserted into the cleaned table.

✅ Final Query

```
-- 26. Cleaning 'shipping_method'
INSERT INTO analytics.retail_clean (shipping_method)
SELECT
   INITCAP(
      TRIM(shipping_method)
   ) AS clean_shipping_method
```

```
FROM staging.retail_raw
WHERE shipping_method IS NOT NULL
  AND TRIM(shipping_method) <> ''
  AND INITCAP(TRIM(shipping_method)) IN ('Express', 'Same-Day', 'Standa
rd');
```

🧾 Logic Breakdown

Step	Function	Purpose
`TRIM()`	Removes extra spaces around values.	
`INITCAP()`	Standardizes capitalization for consistent labeling.	
`IN (...)`	Restricts accepted values to three defined shipping methods.	
`IS NOT NULL / <> ''`	Removes missing or blank entries.	

🧠 Key Learnings

1. **Controlled validation** is crucial — explicitly defining valid categories ensures future data quality.
2. **Casing normalization** improves consistency across categorical dimensions in BI dashboards.
3. Even when data appears “clean,” explicit filters guard against silent data drift.
4. Maintaining **consistent query structure** across columns improves readability and makes debugging easier.

✅ Final Takeaway

> The shipping_method column demonstrated the value of maintaining tight categorical control.
>
> Through consistent casing, whitespace cleanup, and validation filters, I ensured that only standardized and meaningful delivery types were preserved — reinforcing reliable downstream analytics in the retail dataset.

26. Cleaning ‘payment_method

The `payment_method` column records how customers completed transactions — e.g., *Cash*, *Credit Card*, *Debit Card*, or *PayPal*.

Accurate and standardized payment data is critical for reliable financial analysis, ensuring reporting systems categorize transactions correctly.

🧩 Initial Observations

When examining distinct values from `staging.retail_raw`, I identified four main valid payment types:

- `Cash`
- `Credit Card`
- `Debit Card`
- `PayPal`

However, the column contained inconsistencies such as:

- Mixed casing (`credit card`, `PAYPAL`)
- Trailing or leading spaces
- Occasional nulls or blanks

🧠 Cleaning Approach

My cleaning strategy focused on **normalizing formatting** and **enforcing valid categories**:

1. **Casing & Whitespace Normalization**
   - Applied `INITCAP()` to make all values consistently titled.
   - Applied `TRIM()` to remove unnecessary spaces.
2. **Validation Layer**
   - Restricted accepted values using `IN (...)` to prevent invalid entries like `'card'` or `'paypal.com'`.
3. **Data Integrity Filters**
   - Excluded nulls and empty strings for completeness.

✅ Final Query

```
INSERT INTO analytics.retail_clean (payment_method)
SELECT
    INITCAP(
        TRIM(payment_method)
    ) AS clean_payment_method
FROM staging.retail_raw
WHERE payment_method IS NOT NULL
  AND TRIM(payment_method) <> ''
  AND INITCAP(TRIM(payment_method)) IN ('Cash', 'Credit Card', 'Debit Ca
rd', 'PayPal');
```

🧾 Logic Breakdown

Step	Function	Purpose
`TRIM()`	Removes leading and trailing whitespace.	
`INITCAP()`	Standardizes casing for readability and consistency.	
`IN (...)`	Ensures only valid payment types are accepted.	
`IS NOT NULL`	Removes null entries from inclusion.	

💡 Key Learnings

- Enforcing **categorical validation** protects data integrity across financial systems.
- Even clean-looking data benefits from **case and whitespace normalization**.
- Consistent transformation patterns across columns simplify long-term maintainability.

✅ Final Takeaway

> The payment_method column was successfully cleaned using consistent normalization and validation logic.
>
> This guarantees that all entries are clean, standardized, and fall within the correct set of payment options — forming a reliable foundation for financial

> performance analysis and BI reporting.

27. Cleaning ‘order_status’

The `order_status` column represents the fulfillment stage of a customer’s order — whether it’s been processed, shipped, delivered, or is still pending.

Maintaining consistency in this column is critical for order tracking, customer satisfaction metrics, and fulfillment analytics.

🧩 Initial Observations

When profiling the data using:

```
SELECT DISTINCT(order_status) FROM staging.retail_raw;
```

I found four distinct statuses:

- `Delivered`
- `Pending`
- `Processing`
- `Shipped`

Some entries, however, had **inconsistent casing** (e.g. `delivered`, `shipped`) and minor spacing issues.

The goal was to clean and standardize these values while ensuring only valid categories were retained.

🧪 Approach

My approach focused on three key transformations:

1. **Normalization of Casing** using `INITCAP()` to maintain readability and uniformity.
2. **Whitespace Cleanup** with `TRIM()` to remove any leading or trailing spaces.
3. **Category Validation** using `IN (...)` to restrict accepted values to the four known statuses.

✅ Final Query

```
INSERT INTO analytics.retail_clean (order_status)
SELECT
    INITCAP(
        TRIM(order_status)
    )
FROM staging.retail_raw
WHERE order_status IS NOT NULL
  AND TRIM(order_status) <> ''
  AND INITCAP(TRIM(order_status)) IN ('Delivered', 'Pending', 'Processing',
'Shipped');
```

🧾 Logic Breakdown

Step	Function	Purpose
`TRIM()`	Removes unwanted spaces before or after the text.	
`INITCAP()`	Converts text to title case for consistency.	
`IN (...)`	Ensures only valid order statuses are kept.	
`IS NOT NULL`	Filters out missing values.	

🧠 Key Learnings

- Consistent **categorical cleaning patterns** make the ETL pipeline reusable and reliable.
- Explicit validation using `IN` lists guarantees only valid statuses exist in downstream data.
- Clean order status fields support trustworthy metrics like “on-time delivery rate” and “average processing time.”

✅ Final Takeaway

> The order_status column was standardized and validated, ensuring all order lifecycle stages are formatted consistently and contain only approved values.

> This provides a clean foundation for fulfillment and logistics analysis within the retail dataset.

28. Cleaning ‘ratings’

The `ratings` column represents customer satisfaction scores on a scale of 0 to 5, where higher values indicate a better experience. Since this column directly affects performance metrics like *average customer rating* and *feedback distribution*, ensuring both **format consistency** and **numeric accuracy** is critical.

🧩 Initial Observations

Upon exploring distinct values in the `ratings` column:

- Most values were floats ending with `.0` (e.g., `5.0`, `3.0`, `2.0`).
- The column was stored as `TEXT`, not `NUMERIC`, even though it contained numeric-looking values.
- A few entries had unnecessary trailing spaces.
- All values seemed within a valid 0–5 range, but verification was required.

My goal was to:

1. Remove the `.0` artifact.
2. Exclude nulls and blanks.
3. Enforce valid numeric ranges (0–5).
4. Store ratings as **integers**, not strings.

🧪 First Attempt

My initial query focused on formatting:

```
INSERT INTO analytics.retail_clean (ratings)
SELECT
   TRIM(
      REGEXP_REPLACE(ratings::TEXT, '\.0$', '', 'g')
)
```

```
FROM staging.retail_raw
WHERE ratings IS NOT NULL
  AND ratings<> ''
  AND ratings::TEXT ~ '^[0-5]+(\.0)?$';
```

🔍 What Worked

- The regex pattern `'^[0-5]+(\.0)?$'` correctly filtered only numeric values between `0` and `5`, including decimals ending in `.0`.
- `TRIM()` successfully removed any whitespace.
- `.0` artifacts were cleaned effectively using `REGEXP_REPLACE`.

⚠️ What Needed Improvement

- The values were inserted as **text**, not numeric.
- The regex check alone didn't guarantee that numbers were between 0 and 5 — for instance, `55.0` would technically match the pattern.
- I wanted to future-proof the logic by explicitly validating numeric range and typecasting.

🧠 Refined Approach

To align with **industry standards** for numeric data cleaning:

- I explicitly cast the cleaned value to `INTEGER`.
- I added a range validation clause using `BETWEEN 0 AND 5` to reinforce business rules.
- I introduced an alias (`clean_rating`) for better readability.

✅ Final Query

```
-- 29. Cleaning 'ratings'
INSERT INTO analytics.retail_clean (ratings)
SELECT
   CAST(
      REGEXP_REPLACE(ratings::TEXT, '\.0$', '', 'g') AS INTEGER
```

```
    ) AS clean_rating
FROM staging.retail_raw
WHERE ratings IS NOT NULL
  AND TRIM(ratings::TEXT) <> ''
  AND ratings::TEXT ~ '^[0-5]+(\.0)?$'
  AND CAST(REGEXP_REPLACE(ratings::TEXT, '\.0$', '', 'g') AS NUMERIC) B
ETWEEN 0 AND 5;
```

🧾 Logic Breakdown

Step	Function	Purpose
`ratings::TEXT`	Converts numeric-looking values to text for regex handling.	
`REGEXP_REPLACE(..., '\.0$', '', 'g')`	Removes `.0` from float-formatted ratings.	
`CAST(... AS INTEGER)`	Ensures the value is stored as a numeric type.	
`TRIM()`	Cleans any leading/trailing whitespace.	
`ratings::TEXT ~ '^[0-5]+(\.0)?$'`	Filters only valid numeric formats.	
`BETWEEN 0 AND 5`	Enforces valid rating range (business rule).	

💡 Key Learnings

1. **Regex validation** ensures the data format is correct, while **numeric range validation** enforces business logic.
2. Converting to **integer type** is crucial for accurate statistical analysis — text-based numbers can cause misleading aggregations.
3. Maintaining clear aliases and consistent formatting improves readability when scaling cleaning scripts across many columns.
4. Each cleaning step should serve a single purpose — format handling, validation, or type conversion.

✅ Final Takeaway

> The ratings column was successfully cleaned, standardized, and validated to ensure it contained only numeric values between 0 and 5.
>
> Through regex filtering, range validation, and proper typecasting, I ensured this metric could be safely used in future aggregation and customer satisfaction reporting.

29. Cleaning 'products'

The `products` column contains a variety of free-text entries describing individual product names (e.g., *"QLED TV"*, *"Asus ZenBook"*, *"Chocolate bars"*). Because product data directly influences sales trend analyses, category insights, and customer segmentation, ensuring a clean and consistent structure is crucial.

🧩 Initial Observations

On profiling:

- Product names were inconsistently cased (e.g., `"plasma tv"`, `"RUGS"`).
- Some contained trailing punctuation (`"Truffles,"`, `"Asus ZenBook."`).
- Extra spaces were present in a few entries.
- No invalid placeholders were found.

Unlike categorical fields, product names required *format standardization* rather than strict value filtering.

🧪 Cleaning Approach

I decided to:

1. Normalize text casing with `INITCAP()` for readability.
2. Remove trailing punctuation (commas, periods).
3. Replace multiple internal spaces with a single space.
4. Exclude null or empty rows.

✅ Final Query

```
INSERT INTO analytics.retail_clean (products)
SELECT
   INITCAP(
      TRIM(
         REGEXP_REPLACE(
            REGEXP_REPLACE(products, '[,\.]+$', '', 'g'),
            '\s+', ' ', 'g'
         )
      )
   ) AS clean_product
FROM staging.retail_raw
WHERE products IS NOT NULL
  AND TRIM(products) <> '';
```

🧾 Logic Breakdown

Step	Function	Purpose
`REGEXP_REPLACE(... '[,\.]+$' ...)`	Removes punctuation at the end of names.	
`REGEXP_REPLACE(... '\s+' ...)`	Collapses multiple spaces into one.	
`TRIM()`	Removes leading/trailing whitespace.	
`INITCAP()`	Ensures consistent title casing.	
`IS NOT NULL`	Removes empty or missing product entries.	

💡 Key Learnings

- Free-text fields require **format normalization**, not strict categorical validation.
- Over-cleaning can risk removing useful distinctions between product types.
- Consistency in spacing, punctuation, and capitalization improves both data readability and matching accuracy in BI tools.
- This process demonstrates the importance of balancing automation with business understanding.

✅ Final Takeaway

> The products column was standardized for readability and structural consistency without over-normalizing product variations.
>
> This ensured clean, human-readable entries that align with analytical best practices — an essential step in preparing product-level retail data for reporting and modeling.

Initial final insert

```
INSERT INTO analytics.retail_clean (
   transaction_id, customer_id, name, email, phone,
   address, city, state, zipcode, country,
   age, gender, income, customer_segment,
   date, "year", month, time,
   total_purchases, amount, total_amount,
   product_category, product_brand, feedback,
   shipping_method, payment_method, order_status,
   ratings, products
)
SELECT

CAST(transaction_id AS BIGINT)::text

CAST(
      REGEXP_REPLACE(TRIM(customer_id), '\.0[\s\r\n]*$', '')
      AS BIGINT
   )::TEXT

INITCAP(
      TRIM(
         REGEXP_REPLACE(
            REGEXP_REPLACE(name,'^(mr|mrs|ms|dr|miss|prof)[\\.\s]+','','i'),
            '[,\\s]*(phd|md|dds|jr|sr)\\.?$','','i')
         )
```

```
      )

CASE
      WHEN email IS NULL OR TRIM(email) = '' THEN NULL
      WHEN TRIM(email) ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]
{2,}$'
         THEN LOWER(TRIM(email))
      ELSE NULL
   END

TRIM(
      REGEXP_REPLACE(
         REGEXP_REPLACE(phone, '\.0$', ''),
         '[^0-9]', '', 'g'
      )
   )

INITCAP(
      TRIM(
         REGEXP_REPLACE(
            REGEXP_REPLACE(address, '[,\.]+$', '', 'g'),
            '\s+$', '', 'g'
         )
      )
   )

INITCAP(
      TRIM(
         REGEXP_REPLACE(
            REGEXP_REPLACE(
               REGEXP_REPLACE(city, '[,\.]', '', 'g'),
               '\s+$', '', 'g'
            ),
            '\s+', ' ', 'g'
         )
      )
   )
```

```
INITCAP(
      TRIM(
        REGEXP_REPLACE(
          REGEXP_REPLACE(
            REGEXP_REPLACE(state, '[,\.]', '', 'g'),
            '\s+$', '', 'g'
          ),
          '\s+', ' ', 'g'
        )
      )
  )

TRIM(
      REGEXP_REPLACE(
        REGEXP_REPLACE(zipcode::TEXT, '\.0$', '', 'g'),
        '\s+$', '', 'g'
      )
  )

TRIM(UPPER(
      REGEXP_REPLACE(
        REGEXP_REPLACE(
          REGEXP_REPLACE(country, '[,\.]', '', 'g'),
          '\s+$', '', 'g'
        ),
        '\s+', ' ', 'g'
      )
  ))

CAST(
      REGEXP_REPLACE(
        REGEXP_REPLACE(age::TEXT, '\.0$', '', 'g'),
        '\s+$', '', 'g'
      ) AS INTEGER
  )

INITCAP(
      TRIM(
```

```
        REGEXP_REPLACE(gender, '\s+', ' ', 'g')
      )
   )

INITCAP(
      TRIM(
        REGEXP_REPLACE(income, '\s+', ' ', 'g')
      )
   )

INITCAP(
      TRIM(
        REGEXP_REPLACE(customer_segment, '\s+', ' ', 'g')
      )
   )

TO_DATE(TRIM(date), 'MM/DD/YYYY') AS clean_date

CAST(
      TRIM(
        REGEXP_REPLACE("year"::TEXT, '\.0$', '', 'g')
      ) AS INTEGER
)

 INITCAP(
      TRIM(month)
   )

TO_CHAR(
      TO_TIMESTAMP(TRIM(time), 'HH24:MI:SS'),
      'HH24:MI:SS'
   )::TIME

TRIM(
      REGEXP_REPLACE(total_purchases::TEXT, '\.0$', '', 'g')
   )

TRIM(
```

```
        TO_CHAR(ROUND(CAST(amount AS NUMERIC), 2), 'FM999999999.0
0')
    )

TRIM(
        TO_CHAR(ROUND(CAST(total_amount AS NUMERIC), 2), 'FM9999999
99.00')
    )

INITCAP(
        TRIM(product_category)
    )

INITCAP(
        TRIM(
            REGEXP_REPLACE(product_brand, '\s+', ' ', 'g')
        )
    )

INITCAP(
        TRIM(feedback)
    )

INITCAP(
        TRIM(shipping_method)
    )

INITCAP(
        TRIM(payment_method)
    )

INITCAP(
        TRIM(order_status)
    )

CAST(
        REGEXP_REPLACE(ratings::TEXT, '\.0$', '', 'g') AS INTEGER
    )
```

```
INITCAP(
      TRIM(
        REGEXP_REPLACE(
          REGEXP_REPLACE(products, '[,\.]+$', '', 'g'),
          '\s+', ' ', 'g'
        )
      )
   )



FROM staging.retail_raw
WHERE transaction_id, customer_id, name, email, phone,
   address, city, state, zipcode, country,
   age, gender, income, customer_segment,
   date, "year", month, time,
   total_purchases, amount, total_amount,
   product_category, product_brand, feedback,
   shipping_method, payment_method, order_status,
   ratings, products IS NOT NULL
AND LENGTH(REGEXP_REPLACE(phone, '[^0-9]', '', 'g')) BETWEEN 10 AND
15
AND CAST(REGEXP_REPLACE(age::TEXT, '\.0$', '', 'g') AS INTEGER) BETW
EEN 0 AND 120
AND INITCAP(TRIM(gender)) IN ('Male', 'Female')
AND INITCAP(TRIM(income)) IN ('Low', 'Medium', 'High')
AND INITCAP(TRIM(customer_segment)) IN ('New', 'Premium', 'Regular')
AND TO_DATE(TRIM(date), 'MM/DD/YYYY') BETWEEN '1999-01-01' AND '2
025-12-31'
AND INITCAP(TRIM(month)) IN ('January','February','March','April','May','Ju
ne','July','August','September','October','November','December')
AND total_purchases::TEXT ~ '^[0-9]+(\.0)?$'
AND amount::TEXT ~ '^[0-9]+(\.[0-9]+)?$'
AND total_amount::TEXT ~ '^[0-9]+(\.[0-9]+)?$'
AND INITCAP(TRIM(product_category)) IN ('Books','Clothing','Electronic
s','Grocery','Home Decor')
AND INITCAP(TRIM(feedback)) IN('Average', 'Bad', 'Excellent', 'Good')
```

```
AND INITCAP(TRIM(shipping_method)) IN ('Express', 'Same-Day', 'Standar
d')
AND INITCAP(TRIM(payment_method)) IN ('Cash', 'Credit Card', 'Debit Car
d', 'PayPal')
AND INITCAP(TRIM(order_status)) IN ('Delivered', 'Pending', 'Processing',
'Shipped')
AND ratings::TEXT ~ '^[0-5]+(\.0)?$'
AND ratings::TEXT ~ '^[0-5]+(\.0)?$'
AND CAST(REGEXP_REPLACE(ratings::TEXT, '\.0$', '', 'g') AS NUMERIC) BE
TWEEN 0 AND 5;
```

Final Data Load — Combining and Validating All Columns

🧩 Objective

To merge all individually cleaned columns into one unified query and load the final cleaned dataset from the **staging schema** (`staging.retail_raw`) into the **analytics schema** (`analytics.retail_clean`).

This marks the completion of the full SQL cleaning pipeline.

⚙️ Process Overview

After cleaning and testing each column one by one, I built a final **ETL-style SQL script** that:

1. Combines all cleaning logic into a single structured query.
2. Uses a `WITH` CTE to separate **cleaning**, **validation**, and **loading** steps.
3. Ensures every transformation is traceable and consistent with the logic tested earlier.

The final query:

- Cleans all 30 columns simultaneously (regex, trimming, casing, casting).
- Filters invalid or illogical data through a validation layer (`validated` CTE).
- Loads the final clean dataset into `analytics.retail_clean` using a single `INSERT INTO` statement.

🧱 Key Additions

- Added the `product_type` column into both the `cleaned` and `INSERT` stages.

  This ensures every field in the final table now has a matching transformation step.
- Final structure now aligns perfectly with the analytics table schema.
- Consistent naming, indentation, and comments make the SQL easily readable and maintainable.

✅ Outcome

- The cleaned dataset is now fully loaded into `analytics.retail_clean`.
- All 30 columns are properly formatted, validated, and aligned.
- Each value now meets standard analytical data quality checks (no float artifacts, consistent categories, valid ranges, etc.).
- This represents the **final step of the data cleaning phase**.

🧠 Learning Reflection

Building this final query taught me how a real-world ETL pipeline is structured:

- **Modular design** (clean → validate → load)
- **Schema alignment** between staging and analytics
- **Readability and traceability** in SQL scripts

  This approach mirrors how data teams maintain production pipelines, ensuring reliability and auditability across every stage.

🎯 Next Step — Power BI Integration

Now that the data is clean and ready for analysis, the next step is to connect the `analytics.retail_clean` table to **Power BI**.

Using this dataset, I'll begin answering the project's core analytical questions:

1. **Which customer segments generate the highest total purchase value?**

   → Stakeholder: Head of Marketing

   → Visual: Bar or Pie Chart (Revenue by Segment)

2. **What product categories and brands are most popular across months or regions?**

   → Stakeholder: Head of Product

   → Visual: Heatmap or Line Chart (Monthly Trends)

3. **How does average revenue vary by payment or shipping method?**

   → Stakeholder: Head of Finance

   → Visual: Matrix or Comparative Chart

These Power BI visuals will transform the cleaned data into meaningful business insights, completing the **data analytics lifecycle** from raw data → cleaned dataset → interactive dashboard.