Lab: Generate Database Design with ChatGPT

Estimated time needed: 15 minutes

Introduction

In this lab, our primary objective is to empower you with the skills to create a robust and efficient database design using the innovative assistance of ChatGPT. you'll be focusing on the specific task of designing a customer database, and your goal will be to pose insightful questions to ChatGPT regarding the optimal structure and organization of this database.

Learning Objectives

After learning this lab, you should be able to perform the following tasks:

- Create a database using ChatGPT.
- Interact with ChatGPT to seek advice on database design.
- Formulate questions related to database design.

Prologue

Imagine you are tasked with developing a database for a company to manage information about its customers. The key tables you'll be working with are the following:

- Customers: This table is the heart of the database, where you'll store comprehensive information about each customer. Fields such as name, email, and location are crucial for building a solid foundation.

- Locations: To enhance the geographical understanding of your customer base, this table will include details about the customers' locations, encompassing state, country, and zip code.

- Purchases: Tracking the purchases made by each customer is vital for business insights. This table will include details such as the date of purchase, amount spent, and any other relevant information.

Your ultimate goal is to craft a well-structured table design that reflects the interplay between customers, locations, and purchases, ensuring an efficient and coherent database.

As you progress through this lab, your engagement with ChatGPT will guide you in refining your design, and the culmination will be a thoughtfully constructed table design for the customer database. Let's embark on this journey of database design and explore the possibilities that ChatGPT brings to the realm of information organization and management.

> Please note Generative AI is an evolving field. As you attempt the labs, your experience and output might be different than what is seen here.

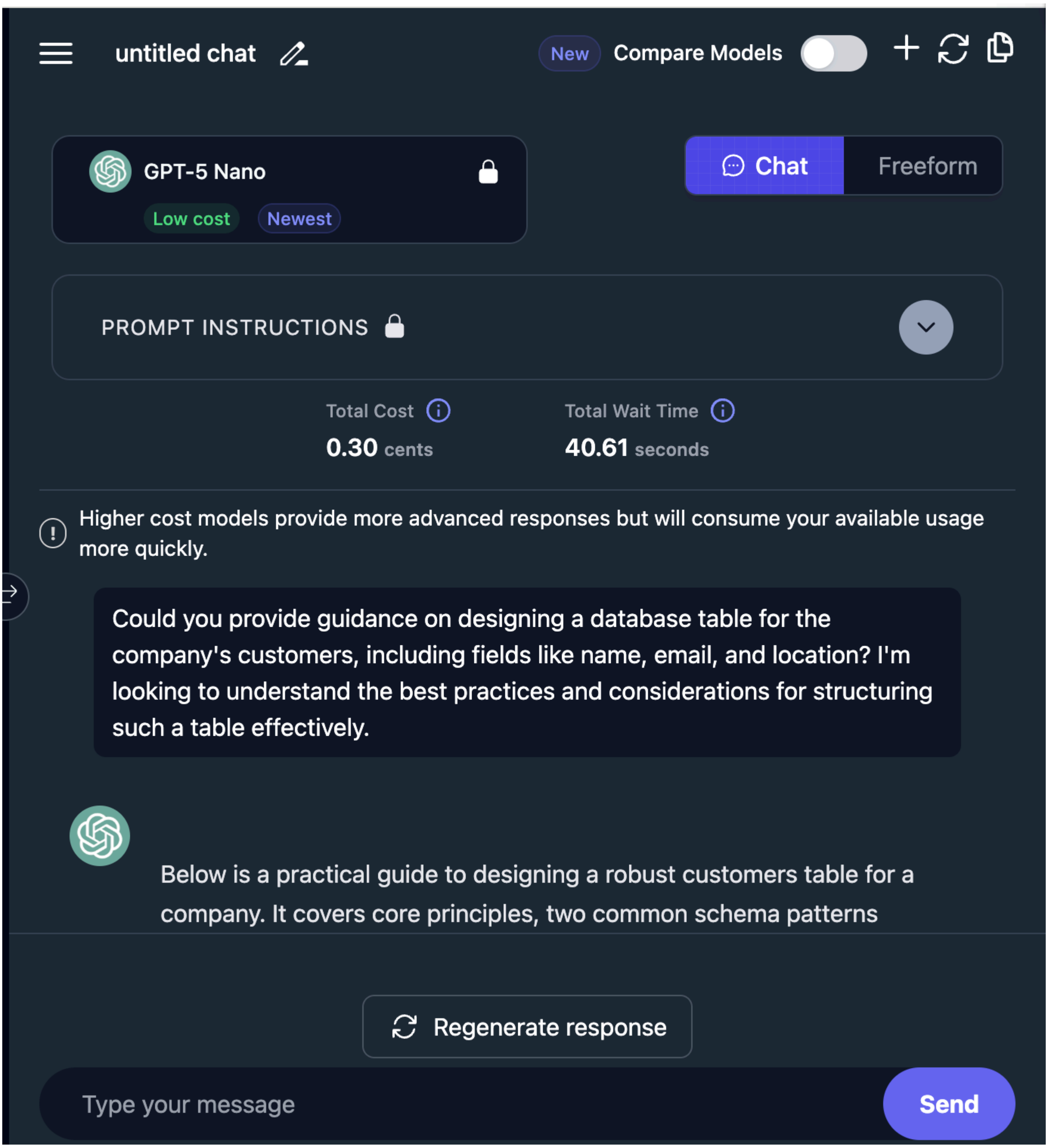
Posing Questions to ChatGPT

In case you need familiarity with the Interface/classroom please see the Lab for reference.

Once you have set up your classroom, let us pose few questions to ChatGPT to create a robust and efficient database design.

1. Please paste the following prompt or question in the **Type your message** field at the bottom of the screen asking ChatGPT to provide you guidance on designing a database table of the company’s customers including fields like name, email, and location.

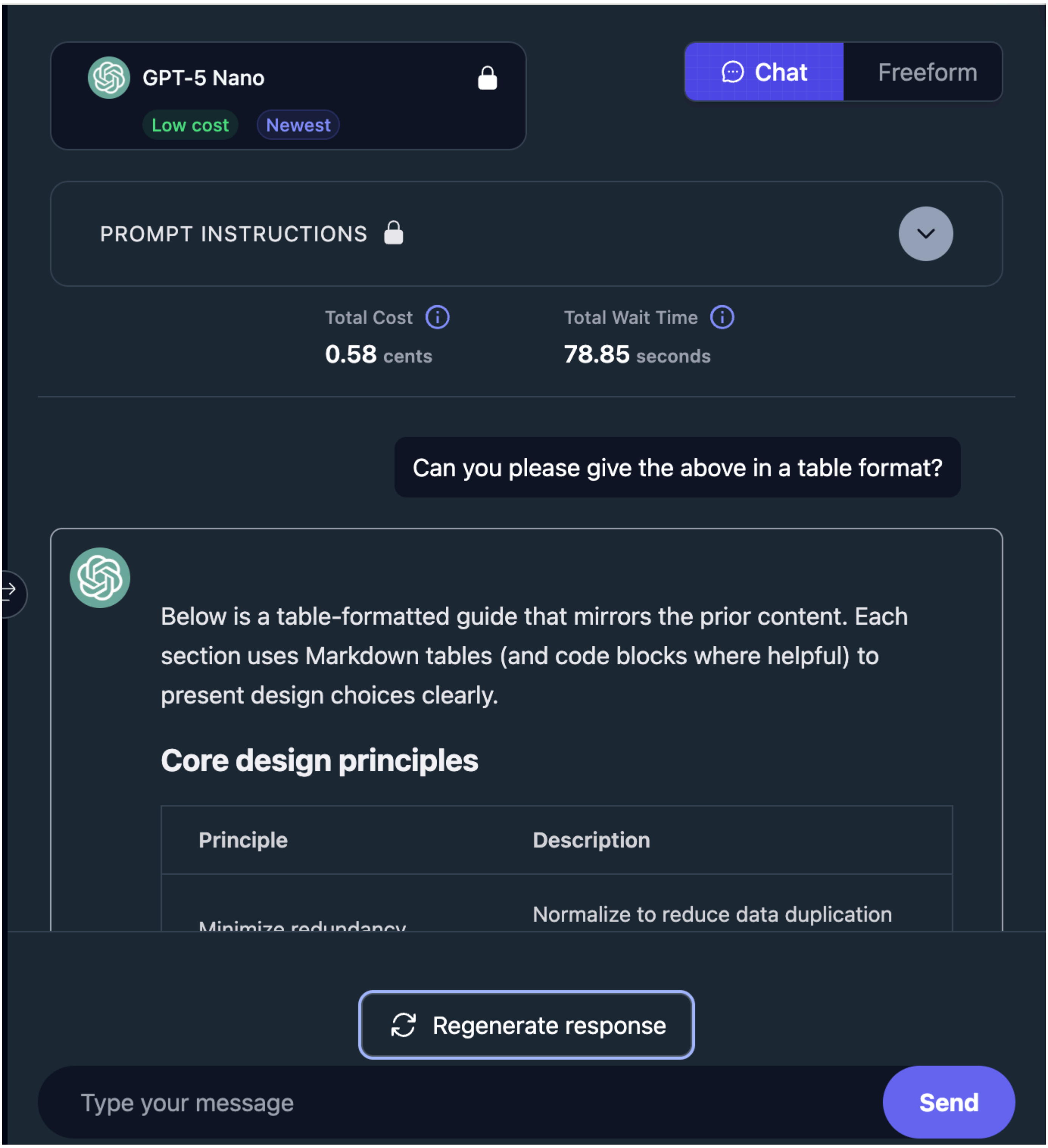
   ```
   Could you provide guidance on designing a database table for the company's customers, including fields like name, email, and location? I'm looking to understand the best practices and considerat
   ```



▶ Click here for full output of the result we got.The output produced at your end may be different.You can also regenerate the response if it is not in the correct format

Then, proceed with the following question:

```
Can you please give the above in a table format?
```

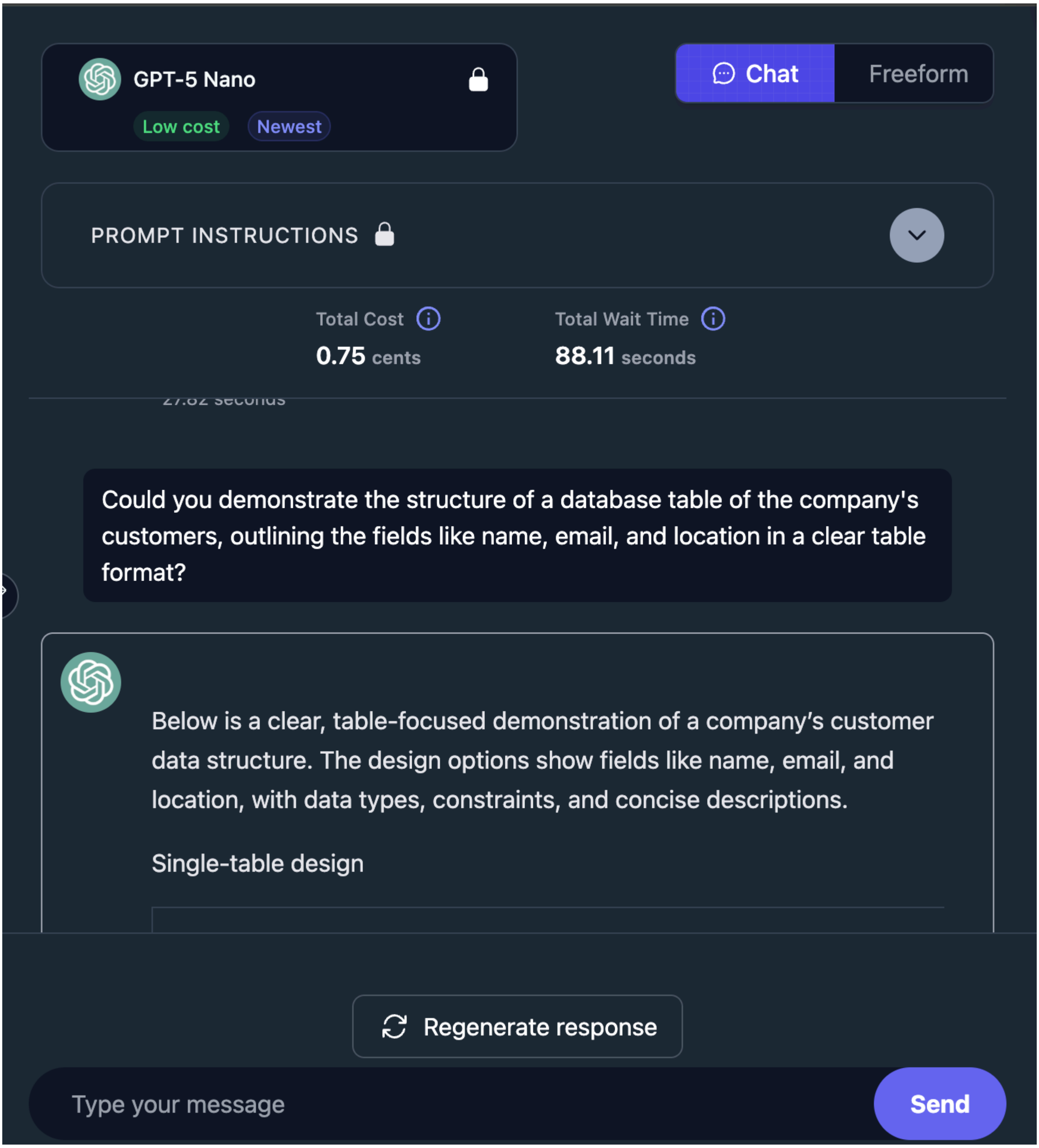


▶ Click here for full output of the result we got.The output produced at your end may be different.

2. Now, let us ask ChatGPT to demonstrate the structure of a database table of the company's customers, outlining fields like name, email, and location.

```
Could you demonstrate the structure of a database table of the company's customers, outlining the fields like name, email, and location in a clear table format?
```

You may see this response:



▼ Click here for full output of the result we got.The output produced at your end may be different.

Below is a clear, table-focused demonstration of a company's customer data structure. The design options show fields like name, email, and location, with data types, constraints, and concise descriptions.

Single-table design

Field	Data type	Constraints	Description
id	BIGINT	PRIMARY KEY	Unique customer identifier
first_name	VARCHAR(100)	NOT NULL	Customer's first name
last_name	VARCHAR(100)	NOT NULL	Customer's last name
email	VARCHAR(254)	NOT NULL, UNIQUE	Customer email address
phone	VARCHAR(32)		Contact number
address_line1	VARCHAR(128)		Address line 1
address_line2	VARCHAR(128)		Address line 2
city	VARCHAR(100)		City
state	VARCHAR(100)		State/Region
postal_code	VARCHAR(20)		Postal code

Field	Data type	Constraints	Description
country_code	CHAR(2)	NOT NULL	ISO country code
status	VARCHAR(20)	NOT NULL DEFAULT ‘active’	Account status
marketing_opt_in	BOOLEAN	NOT NULL DEFAULT FALSE	Marketing consent
created_at	TIMESTAMP WITHOUT TIME ZONE	NOT NULL DEFAULT now()	Record creation time
updated_at	TIMESTAMP WITHOUT TIME ZONE	NOT NULL DEFAULT now()	Last update time
last_contacted_at	TIMESTAMP WITHOUT TIME ZONE		Last outreach time

Notes:
This single-table approach is simple and fast for small to medium datasets but may duplicate location data across records.

Normalized design (locations separated)

Locations table

Field	Data type	Constraints	Description
id	BIGINT	PRIMARY KEY	Unique location identifier
city	VARCHAR(100)		City
state	VARCHAR(100)		State/Region
country_code	CHAR(2)	NOT NULL	ISO country code
postal_code	VARCHAR(20)		Postal code
latitude	NUMERIC(9,6)		Latitude
longitude	NUMERIC(9,6)		Longitude

Constraint: UNIQUE(city, state, country_code, postal_code)

Customers table

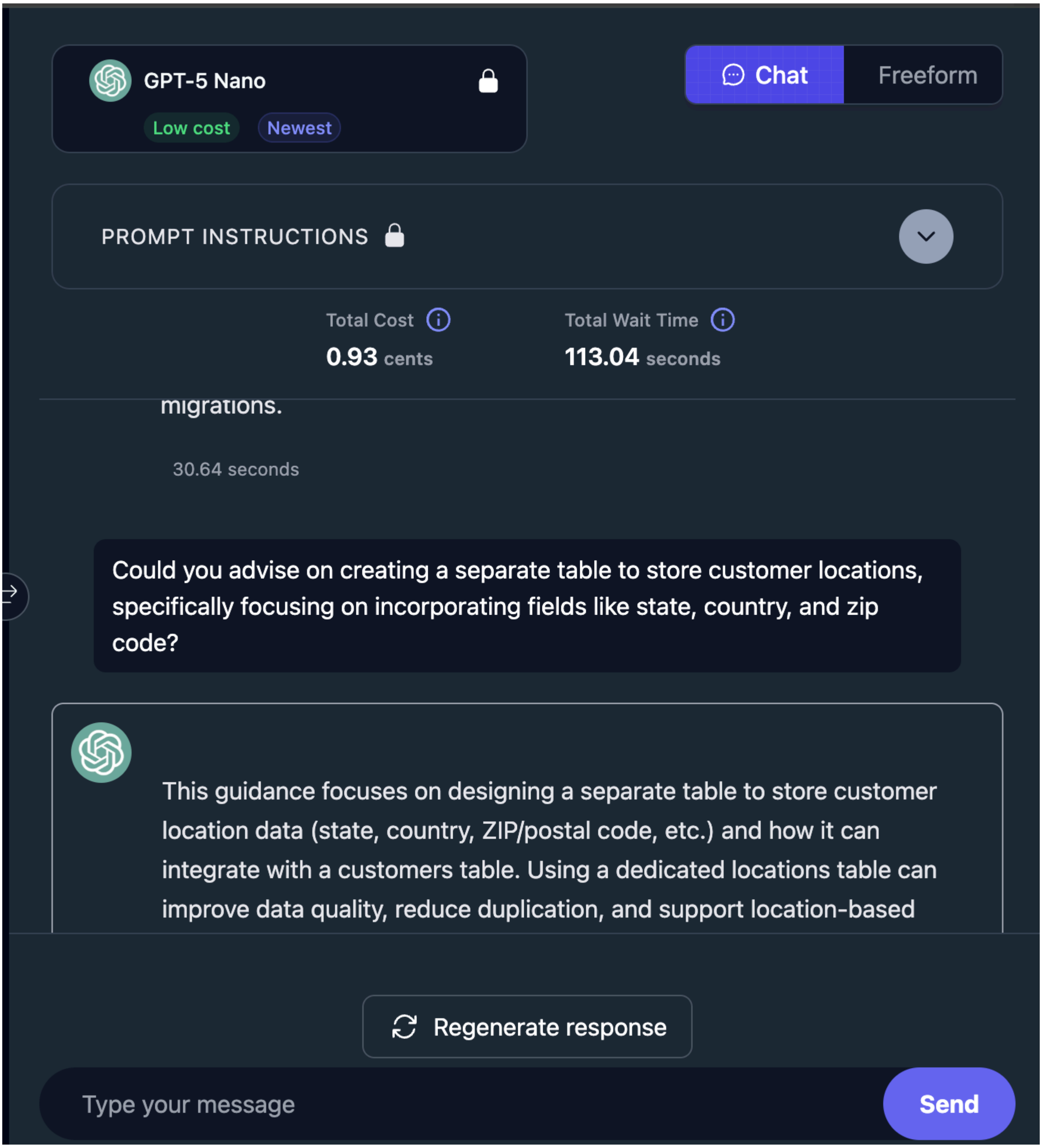
Field	Data type	Constraints	Description
id	BIGINT	PRIMARY KEY	Unique customer identifier
first_name	VARCHAR(100)	NOT NULL	Customer’s first name
last_name	VARCHAR(100)	NOT NULL	Customer’s last name
email	VARCHAR(254)	NOT NULL, UNIQUE	Customer email address
phone	VARCHAR(32)		Contact number
location_id	BIGINT	FOREIGN KEY REFERENCES locations(id)	Reference to location
status	VARCHAR(20)	NOT NULL DEFAULT ‘active’	Account status
marketing_opt_in	BOOLEAN	NOT NULL DEFAULT FALSE	Marketing consent
created_at	TIMESTAMP WITHOUT TIME ZONE	NOT NULL DEFAULT now()	Record creation time
updated_at	TIMESTAMP WITHOUT TIME ZONE	NOT NULL DEFAULT now()	Last update time
last_contacted_at	TIMESTAMP WITHOUT TIME ZONE		Last outreach time

Notes:
The normalized design reduces location data duplication and improves data quality, at the cost of joins and slightly more complex migrations.

3. Next, let us ask ChatGPT to advise you on creating a separate table to store customer locations, focusing on incorporating fields like state, country, and zip code.

```
Could you advise on creating a separate table to store customer locations, specifically focusing on incorporating fields like state, country, and zip code?
```

You may see the following response:



▼ Click here for full output of the result we got.The output produced at your end may be different.

This guidance focuses on designing a separate table to store customer location data (state, country, ZIP/postal code, etc.) and how it can integrate with a customers table. Using a dedicated locations table can improve data quality, reduce duplication, and support location-based analytics. Below are practical options, schema sketches, constraints, indexing tips, and migration considerations.

Why a separate locations table?

Benefit	Description
Normalization	Avoids duplicating address components across many customers.
Reusability	Many customers share the same location (city/state/country/postal code).
Data quality	Centralizes validation and standardization for location fields (country codes, state codes, postal formats).
Extensibility	Easy to add metadata (time zone, geocoordinates, region classifications) without expanding the main customers table.
Analytics	Simplified grouping and filtering by location, with cleaner joins.
Privacy	Location data can be treated according to its sensitivity, with appropriate access controls and retention policies.

Recommended location schema options

Below are two common approaches. Choose based on your domain, data quality needs, and query patterns.

Integrating with the customers table

Typically, the `customers` table references a `location_id`. This keeps customer records lean and makes location updates centralized.

Example DDL (continuing from Option A):

```
CREATE TABLE customers (
  id BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  first_name VARCHAR(100) NOT NULL,
  last_name VARCHAR(100) NOT NULL,
  email VARCHAR(254) NOT NULL UNIQUE,
  location_id BIGINT NOT NULL,
  status VARCHAR(20) NOT NULL DEFAULT 'active',
  marketing_opt_in BOOLEAN NOT NULL DEFAULT FALSE,
  created_at TIMESTAMP WITHOUT TIME ZONE NOT NULL DEFAULT now(),
  updated_at TIMESTAMP WITHOUT TIME ZONE NOT NULL DEFAULT now(),
  last_contacted_at TIMESTAMP WITHOUT TIME ZONE,
  FOREIGN KEY (location_id) REFERENCES locations(id)
);
```

Notes:

- Enforce NOT NULL on location_id if every customer must have a location; otherwise allow NULL and handle accordingly.
- If some customers have multiple addresses, consider a separate customer_locations table (many-to-many or one-to-many if applicable).
- For the Option B approach, location_id would reference locations(id) in the normalized locations table, with similar migrations applied.

Practical constraints and data quality considerations

Area	Recommendation
Country codes	Use a constrained, standardized format (e.g., ISO codes). Consider a countries reference table or a CHECK constraint.
State/province	Normalize state/province codes where feasible; provide a states reference table if you require strict validation.
Required fields	Decide on required vs optional fields per country. For example, postal_code is essential for many regions but may be optional in others.
Geospatial data	Consider storing latitude/longitude for geospatial queries and time zone for regional operations.
Uniqueness	Enforce uniqueness where appropriate (e.g., combination of country_code, state_code, city, postal_code) to avoid duplicates, especially in the locations table.
Audit fields	Include audit fields (created_at, updated_at) and consider an updated_by column for traceability.

Indexes and performance guidance

Area	Guidance
Joins	Index location lookups to speed joins with customers: location_id.
Customer filtering	Index frequently filtered fields in customers (email, status, created_at) as appropriate for your workload.
Location filtering	If you frequently query by country/state, consider indexes on country_code and (country_code, state_code).
Large datasets	Consider partitioning by country or by region and/or using geographic indexes if supported by your DBMS.
Case-insensitive search	If using a languages-agnostic case-insensitive search on country/state names, consider functional indexes (where supported).

Migration and evolution tips

- Start with a simple locations table anchored by country_code and optional city/state/postal_code fields.
- If moving toward a strongly normalized model, plan a two-phase migration:
  1. Populate countries/states references
  2. Migrate customers to reference location_id, then deprecate inline address fields.
- Use versioned migrations and test data integrity thoroughly in a staging environment before production changes.
- Consider feature flags to roll out the normalization incrementally and rollback if needed.

Quick example queries

Find all customers in a given country:

```
SELECT c.id, c.first_name, c.last_name, c.email
FROM customers c
JOIN locations l ON c.location_id = l.id
WHERE l.country_code = 'US';
```

Get counts by city within a country:

```
SELECT l.city, COUNT(*) AS customer_count
FROM customers c
JOIN locations l ON c.location_id = l.id
WHERE l.country_code = 'US'
GROUP BY l.city
ORDER BY customer_count DESC;
```

Update a location's postal code (with care for normalization):

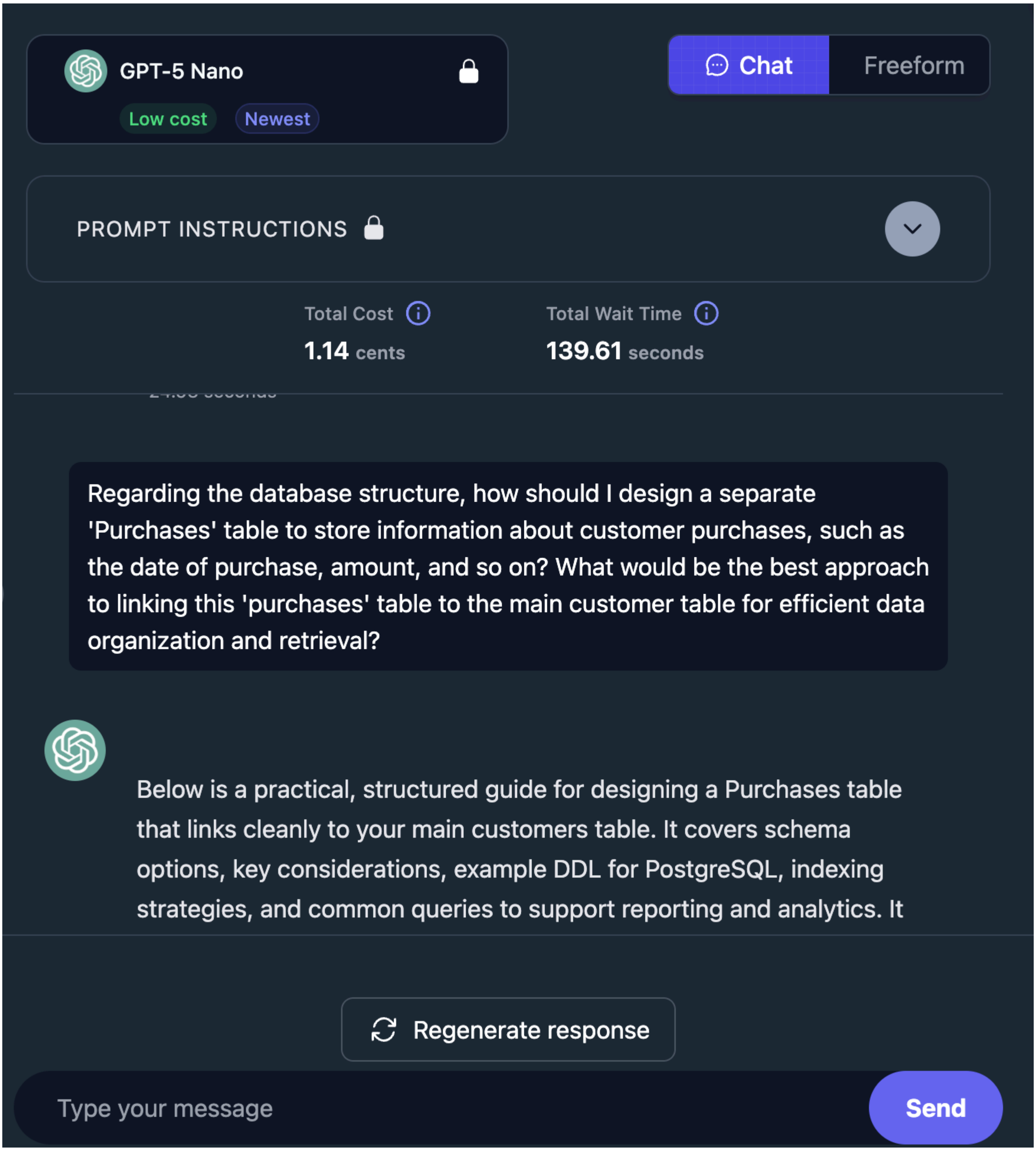
```
UPDATE locations
```

```
SET postal_code = '94107', updated_at = NOW()
WHERE id = :location_id;
```

4. Let us now enquire about how we can design a separate **Purchases** table to store information about customer purchases, such as the date of purchase, amount, and so on, and what would be the best approach to linking this **purchases** table to the main customer table for efficient data organization and retrieval.

```
Regarding the database structure, how should I design a separate 'Purchases' table to store information about customer purchases, such as the date of purchase, amount, and so on? What would be t
```

You may see this response:



▼ Click here for full output of the result we got.The output produced at your end may be different.

Below is a practical, structured guide for designing a Purchases table that links cleanly to your main customers table. It covers schema options, key considerations, example DDL for PostgreSQL, indexing strategies, and common queries to support reporting and analytics. It also addresses data privacy and migration concerns when evolving an existing schema.

Overview and design goals

Goals:

- Capture each purchase with essential details (date, amount, currency, status, payment method).
- Link purchases to customers via a foreign key for traceability and analytics.
- Accommodate future needs with an extensible design (line items, tax/shipping, discounts).

Key considerations:

- Avoid storing sensitive payment card data; store tokens or references from a PCI-compliant gateway and last4 digits if needed.
- Decide between a simple one-row-per-purchase table and a full order-line model depending on whether you need line-item granularity in queries.
- Plan for growth: indexing, partitioning by date, and potential historical/archive strategies.

Schema options

Option A (simple, one row per purchase):
One purchase record per event; suitable for straightforward revenue tracking.

Option B (purchases with line items):
Supports multiple items per purchase; better for detailed analytics but requires a second table for line items.

Option A: Simple purchases table

In this design, each row represents a single purchase. It stores the total amount for the order, currency, and a link to the customer.

Key fields (example):

- id: primary key
- customer_id: foreign key to customers(id)
- purchase_date: timestamp
- amount: numeric(12,2)
- currency: ISO 4217 code (3-letter)
- status: e.g., 'completed', 'pending', 'refunded'
- payment_reference: string token or gateway reference
- shipping_cost: numeric(12,2)
- tax_amount: numeric(12,2)
- created_at / updated_at: auditing timestamps

Option B: Purchases with line items (order-level with items)

This approach adds a purchase_items table to capture each item in an order. It enables detailed revenue per product and better inventory alignment.

Core tables:

- purchases: (id, customer_id, purchase_date, total_amount, currency, status, payment_reference, shipping_cost, tax_amount, created_at, updated_at)
- purchase_items: (id, purchase_id, product_id, quantity, unit_price, line_total)

Notes:

- If you have a Products table, link via product_id. If not, consider name/description fields in purchase_items for item details.
- You can add constraints like CHECK (quantity > 0) and (line_total = quantity * unit_price) where supported by your DBMS.

PostgreSQL DDL examples

Option A: Simple purchases table

```
-- Purchases (one row per purchase)
CREATE TABLE purchases (
  id BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  customer_id BIGINT NOT NULL,
  purchase_date TIMESTAMP WITHOUT TIME ZONE NOT NULL DEFAULT now(),
  amount NUMERIC(12,2) NOT NULL,
  currency CHAR(3) NOT NULL DEFAULT 'USD',
  status VARCHAR(20) NOT NULL DEFAULT 'completed',
  payment_reference VARCHAR(100),
  shipping_cost NUMERIC(12,2) DEFAULT 0,
  tax_amount NUMERIC(12,2) DEFAULT 0,
  created_at TIMESTAMP WITHOUT TIME ZONE NOT NULL DEFAULT now(),
  updated_at TIMESTAMP WITHOUT TIME ZONE NOT NULL DEFAULT now(),
  FOREIGN KEY (customer_id) REFERENCES customers(id) ON DELETE RESTRICT
);
-- Optional: index for efficient lookups by customer and date
CREATE INDEX idx_purchases_customer_date ON purchases (customer_id, purchase_date);
```

Option B: Purchases with line items

```
-- Purchases table (order header)
CREATE TABLE purchases (
  id BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  customer_id BIGINT NOT NULL,
  purchase_date TIMESTAMP WITHOUT TIME ZONE NOT NULL DEFAULT now(),
  total_amount NUMERIC(12,2) NOT NULL,
  currency CHAR(3) NOT NULL DEFAULT 'USD',
  status VARCHAR(20) NOT NULL DEFAULT 'completed',
  payment_reference VARCHAR(100),
  shipping_cost NUMERIC(12,2) DEFAULT 0,
  tax_amount NUMERIC(12,2) DEFAULT 0,
  created_at TIMESTAMP WITHOUT TIME ZONE NOT NULL DEFAULT now(),
  updated_at TIMESTAMP WITHOUT TIME ZONE NOT NULL DEFAULT now(),
  FOREIGN KEY (customer_id) REFERENCES customers(id) ON DELETE RESTRICT
);
-- Purchase line items (one row per item in the order)
CREATE TABLE purchase_items (
  id BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  purchase_id BIGINT NOT NULL,
  product_id BIGINT,
  description VARCHAR(255),
  quantity INTEGER NOT NULL CHECK (quantity > 0),
  unit_price NUMERIC(12,2) NOT NULL CHECK (unit_price >= 0),
  line_total NUMERIC(12,2) NOT NULL CHECK (line_total >= 0),
  FOREIGN KEY (purchase_id) REFERENCES purchases(id) ON DELETE CASCADE
  -- Optional: FOREIGN KEY (product_id) REFERENCES products(id)
```

```
);
-- Ensure total mechanics (optional, for data integrity):
-- ALTER TABLE purchase_items ADD CONSTRAINT chk_line_total CHECK (line_total = quantity * unit_price);
```

Notes:

- In Option B, you maintain a total at the header (total_amount) and each line item has its own quantity and price.
- Including a derived column like line_total can help reporting but may require triggers or application logic to keep in sync with unit_price and quantity.
- For real systems, you'll likely have a products or catalog table; reference product_id there if available.

Example queries (typical use cases)

Get all purchases for a given customer (recent first):

```
SELECT p.*
FROM purchases p
WHERE p.customer_id = :customer_id
ORDER BY p.purchase_date DESC;
```

Total spend by customer over a period:

```
SELECT c.id AS customer_id, SUM(p.amount) AS total_spent
FROM purchases p
JOIN customers c ON p.customer_id = c.id
WHERE p.purchase_date BETWEEN :start_date AND :end_date
GROUP BY c.id
ORDER BY total_spent DESC;
```

If using line items (Option B): total revenue per product across all purchases:

```
SELECT pi.product_id, SUM(pi.line_total) AS revenue
FROM purchase_items pi
JOIN purchases p ON pi.purchase_id = p.id
WHERE p.purchase_date >= :start_date
GROUP BY pi.product_id
ORDER BY revenue DESC;
```

Average order value (AOV) per customer:

```
SELECT c.id, AVG(p.amount) AS average_order_value
FROM purchases p
JOIN customers c ON p.customer_id = c.id
GROUP BY c.id;
```

Last purchase date per customer:

```
SELECT c.id, MAX(p.purchase_date) AS last_purchase
FROM purchases p
JOIN customers c ON p.customer_id = c.id
GROUP BY c.id;
```

Indexes, partitioning, and performance tips

- Create an index on (customer_id, purchase_date) to optimize customer-centric and date-range queries.
- Consider a separate index on (purchase_date) for time-based analytics.
- If you have very large datasets, consider partitioning purchases by purchase_date (e.g., by year or quarter) to improve maintenance and query performance.
- If you frequently query by currency, add an index on currency (and possibly a composite index on (currency, purchase_date)).
- For Option B, index purchase_items.purchase_id to speed up joins and item-level aggregations.
- Validate data redundancies with constraints: CHECK (amount >= 0), CHECK (tax_amount >= 0), CHECK (shipping_cost >= 0).
- Consider soft-delete or archiving for older purchases if regulatory or business requirements demand historical retention without cluttering hot storage.

Data privacy, security, and compliance considerations

- Do not store full payment card data; store references to a payment gateway and, if needed for business reasons, the last4 digits and transaction IDs only.
- Use least privilege RBAC to restrict access to purchase data; separate duties for billing, support, and analytics.
- Keep audit fields (created_at, updated_at, potentially updated_by) for accountability.
- If you must keep historical customer data after deletion, consider soft-delete patterns (e.g., a deleted_at column) and ensure reports exclude logically deleted rows.
- Consider data retention policies and regulatory requirements (GDPR, CCPA) when designing deletion or anonymization processes.

Migration and evolution tips

- If adopting Option B after starting with Option A, plan a two-step migration:
  1. Add a purchases table linked to customers and populate with historical purchases (if available).
  2. Introduce purchase_items and migrate line items as needed.
- Use versioned migrations and test thoroughly in a staging environment before applying to production.
- For reporting, consider exporting purchase data to a data warehouse or analytics store to avoid impacting OLTP performance.
- Implement data quality checks (e.g., sums and totals are consistent) during migrations.

Summary

Through this hands-on experience, you've explored the dynamic synergy between your creative insights and the innovative assistance provided by ChatGPT in crafting a robust customer database.

In your journey, you posed insightful questions, navigated the intricacies of structuring the Customers, Locations, and Purchases tables, and collaborated with ChatGPT to refine your design. The result is a thoughtfully constructed database that lays the groundwork for efficient information organization and management.

This lab not only honed your skills in database design but also showcased the potential of leveraging AI tools like ChatGPT in real-world scenarios. As you reflect on your experience, consider the valuable lessons learned, challenges overcome, and the collaborative spirit fostered in the pursuit of an optimal database design.

Congratulations!

You have completed the lab on generating database design with ChatGPT.

Author(s)

Dania Kulsum