



دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

پایگاه داده پیشرفته - مقطع کارشناسی ارشد

گزارش پروژه پایانی پایگاه داده پیشرفته

پلتفرم هوشمند تحلیل داده‌های شرکت Uber

حسین مستاجران گورтанی - ۴۰۴۳۶۱۴۰۳۳

لینک ریپازیتوری:

<https://github.com/HosseinMst81/uber-data-platform/>

۱۴۰۴ بهمن

چکیده

این پروژه پایانی درس پایگاه داده پیشرفته، طراحی و پیاده‌سازی یک پلتفرم هوشمند تحلیل داده‌های تراکنش‌های شرکت Uber در سال ۲۰۲۴ را با استفاده از معماری (Gold, Silver, Bronze) Medallion به انجام رساند. ابتدا داده‌های خام با ۱۵۰,۰۰۰ رکورد بدون هیچ تغییری در لایه Bronze ذخیره شد. سپس در لایه Silver با انجام عملیات ETL، مهندسی ویژگی (اضافه کردن datetime و ستون‌های زمانی)، یکپارچه‌سازی دلایل لغو سفر در یک ستون واحد و مدیریت هوشمند مقادیر Null (به‌ویژه پر کردن میانگین امتیازها در سفرهای تکمیل شده) داده‌ها تمیز و آماده تحلیل شد. در لایه Gold، جدولنهایی با کلید اصلی، محدودیت‌های CHECK روی امتیازها و محاسبه متريک کلیدی revenue_per_km ساخته شد تا داده‌ها برای گزارش‌گیری و داشبورد بهینه باشند.

یک API ساده CRUD روی لایه Gold پیاده‌سازی شد تا امکان ایجاد، مشاهده (با فیلتر customer_id)، ویرایش وضعیت و حذف سفرها فراهم شود. داشبورد تعاملی با شاخص‌های کلیدی عملکرد (تعداد سفرها، درآمد کل، نرخ موفقیت، توزیع لغوها و ...) و نمودارهای متنوع (Line, Bar, Pie) با قابلیت فیلتر زمانی و نوع خودرو طراحی گردید. در بخش دستیار هوشمند Text-to-SQL، با استفاده از مدل Llama از طریق OpenRouter و پرامپت مهندسی شده شامل schema جدول، سوالات طبیعی به کوئری SELECT آمن تبدیل می‌شوند. در نهایت با ایجاد ایندکس‌های مناسب و مقایسه زمان اجرا با EXPLAIN ANALYZE، بهبود عملکرد دیتابیس سنجیده شد و جستجوی معنایی روی دلایل لغو با Chroma DB پیاده‌سازی گردید.

این سیستم ترکیبی از مفاهیم پایگاه داده پیشرفته، مهندسی داده و هوش مصنوعی را در یک پلتفرم کاربردی و کامل به نمایش گذاشت.

صفحه

فهرست مطالب

۷	مرحله اول: آشنایی با دیتاست
۷	- مرحله دوم: طراحی معماری پایه بر اساس Bronze Medallion Architecture
۸	- مرحله سوم: طراحی معماری پایه بر اساس Silver Medallion Architecture
۹	- مرحله چهارم: طراحی معماری پایه بر اساس Gold Architecture Medallion
۹	- مرحله پنجم: پیاده سازی CRUD API
۹	- مرحله ششم: داشبور تحلیلی و تعاملی
۱۰	- مرحله هفتم: دستیار هوشمند Text to SQL
۱۰	- مرحله هشتم: بهینه سازی و ایندکس گذاری
۱۱	- مرحله نهم: پیاده سازی Vector Database و جستجوی معنایی (Semantic Search)
۱۲	پیاده سازی پروژه
۱۳	گزارش فاز اول: تحلیل و شناسایی مجموعه داده (Data Analysis)
۱۳	- توصیف کلی (General Description)
۱۴	- تحلیل ستون ها و روابط بین آن ها
۱۸	گزارش فاز دوم: Bronze لایه Medallion Architecture
۲۲	گزارش فاز سوم: Silver لایه Medallion Architecture
۲۳	- راه اندازی اولیه تراکنش
۲۴	- اقدام اول مهندسی ویژگی (Feature Engineering)
۲۵	- پاکسازی داده های قبلي و درج اولیه از برنز
۲۸	- اقدام دوم: یکپارچه سازی دلایل لغو سفر
۲۸	- اقدام سوم: مدیریت مقادیر Null
۲۹	- اقدام چهارم: تحلیل و مدیریت امتیازها
۳۱	- پایان تراکنش و لاغ نهایی
۳۶	- 6- بررسی تعداد رکوردها و مدیریت داده های تکراری
۳۸	گزارش فاز چهارم: Gold لایه Medallion Architecture
۴۲	گزارش فاز پنجم: پیاده سازی CRUD API
۴۲	- ساختار تعریف شده Backend
۴۴	- 2-2- عملیات CRUD
۴۵	- 2-3-1- CREATE - Post API

۴۶ READ - GET API 2-3-2-
۴۸ UPDATE - PATCH API 2-3-3-
۴۹ DELETE API - ۲-۳-۴
۵۰	گزارش فاز ششم : داشبورد تحلیلی و تعاملی
۵۰	2-۱- طراحی معماری سمت بکاند برای داشبورد
۵۱	2-۲- مدیریت ایمن مقادیر عددی
۵۲	2-۳- شاخص‌های کلیدی عملکرد(KPI)
۵۴	2-۴- نمودارها(Charts)
۵۴	2-۳-۱- نمودار دایره‌ای توزیع دلایل لغو.
۵۶	2-۳-۲- نمودار توزیع روش‌های پرداخت
۵۷	2-۳-۳- نمودار مقایسه نوع خودرو
۵۸	2-۳-۴- نمودار ساعات پرتردد
۵۹	2-۳-۵- نمودار تحلیل روزهای هفتگی
۶۱	گزارش فاز هفتم: دستیار هوشمند Text to SQL
۶۱	2-۱- پیاده‌سازی بکاند
۶۵	گزارش فاز هشتم: بهینه‌سازی و ایندکس‌گذاری
۶۵	2-۱- انتخاب کوئری سنگین
۶۶	2-۲- اجرای کوئری قبل از ایندکس‌گذاری
۶۸	2-۳- ایجاد ایندکس‌های مناسب
۶۹	2-۴- خروجی کوئری با استفاده از ایندکس
۷۰	2-۵- مقایسه
۷۲	منابع و مراجع
۷۴	پیوست‌ها:

صفحه

فهرست اشکال

۱۷ شکل ۱ دیاگرام موجودیت‌ها و روابط (ER Diagram)
۱۸ شکل ۲ راهاندازی اولیه PostgreSQL از طریق Shell psql
۱۹ شکل ۳ ساختن اسکیمای Bronze و ساخت جدول لایه اول
۲۰ شکل ۴ ارور وجود null رشته‌ای در فایل csv
۲۱ شکل ۵ کد وارد کردن داده‌های خام از فایل csv به داخل bronze schema و جدول raw_dataset
۲۱ شکل ۶ چک کردن اتصال به PostgreSQL.
۲۲ شکل ۷ اسکیمای Bronz و جداول ساخته شده در PostgreSQL DB
۲۲ شکل ۸ لایه Bronze و raw_dataset در PostgreSQL DB
۳۱ شکل ۹ اجرای موفق کد silver_etl و لایه silver
۳۱ شکل ۱۰ ساختار جدول Bronze
۳۲ شکل ۱۱ ساختار جدول Silver
۳۲ شکل ۱۲ داده‌های لایه برنز قبل از تغییرات در pgAdmin4
۳۳ شکل ۱۳ داده‌ها در لایه Silver بعد از تغییرات در pgAdmin4
۳۳ شکل ۱۴ مقایسه null value ها در دو لایه نقره و برنز و درصد بهبود
۳۴ شکل ۱۵ جدول Silver و مدیریت مقادیر Null
۳۴ شکل ۱۶ عدم وجود Null در ستون جدید unified_cancellation_reason
۳۵ شکل ۱۷ دلایل لغو سفر و تعداد هر کدام به صورت یکپارچه
۳۶ شکل ۱۸ میانگین امتیازها با توجه به نوع خودرو
۳۷ شکل ۱۹ بررسی وجود booking_id تکراری
۳۷ شکل ۲۰ مقایسه تعداد رکوردهای جدول Silver و Bronze
۳۹ شکل ۲۱ خروجی موفق عملیات Gold
۳۹ شکل ۲۲ ساختار و اسکیمای جدول Gold
۴۰ شکل ۲۳ میانگین درامد در هر کلیومتر به تفکیک نوع وسیله نقلیه
۴۱ شکل ۲۴ جدول Gold
۴۵ شکل ۲۵ - ساخت یک سفر جدید
۴۶ شکل ۲۶ READ - خواندن یک رکورد با customer_id مشخص
۴۷ شکل ۲۷ READ - خواندن همه رکوردها یا اعمال فیلتر با کمک limit و offset
۴۷ شکل ۲۸ READ - خواندن همه رکوردها بدون هیچ فیلتری
۴۸ شکل ۲۹ UPDATE - بروزرسانی وضعیت یک سفر از طریق booking_id
۴۹ شکل ۳۰ DELETE - حذف یک سفر

۵۲	شکل ۳۱ کد تابع safeNumber برای مدیریت مقادیر عددی
۵۳	شکل ۳۲ نمایش KPI ها در داشبورد
۵۳	شکل ۳۳ نمایش KPI ها با فیلتر نوع خودرو
۵۳	شکل ۳۴ نمایش KPI ها با فیلتر زمانی
۵۴	شکل ۳۵ کد مربوط به سفرهای لغو شده
۵۵	شکل ۳۶ نمودار دلایل لغو سفر
۵۶	شکل ۳۷ نمودار دایره‌ای روش‌های پرداخت
۵۸	شکل ۳۸ نمودار میله‌ای مقایسه انواع خودرو
۵۹	شکل ۳۹ نمودار خطی ساعات پرتردد
۶۰	شکل ۴۰ نمودار میله‌ای تعداد سفرها براساس روز هفته
۶۱	شکل ۴۱ توصیف GOLD_SCHEMA برای AI در کنترل
۶۲	شکل ۴۲ تعریف محدودیت‌ها و اعتبارسنجی به صورت Prompt Engineering
۶۳	شکل ۴۳ دستیار هوشمند Text to SQL
۶۴	شکل ۴۴ سوال پیچیده در دستیار هوشمند
۶۴	شکل ۴۵ سوال نامربوط در دستیار هوشمند
۶۶	شکل ۴۶ اجرای کوئری سنگین بدون ایندکس گذاری
۶۷	شکل ۴۷ خروجی کوئری سنگین بدون ایندکس گذاری
۶۹	شکل ۴۸ خروجی کوئری سنگین با استفاده از ایندکس ما

صفحه

فهرست جداول

- ۱۶ جدول ۱ معرفی ستون ها و نوع داده ها (Featuring & Data Types)
- ۴۱ جدول ۲ مقایسه سه لایه معماری مدل ایون
- ۷۰ جدول ۳ مقایسه استفاده از ایندکس و عدم استفاده از ایندکس مناسب

مرحله اول: آشنایی با دیتاست

تحلیل زیرساختی و درک ماهیت داده‌ها: در گام نخست، تمرکز اصلی بر شناخت دقیق دیتاست واقعی تراکنش‌های شرکت Uber است که شامل بیش از ۱۴۸ هزار رکورد است.

قبل از هرگونه طراحی پایگاه داده، لازم است تا ساختار داده‌ها را با دقت بررسی کنید. لطفاً موارد زیر را انجام دهید:

توصیف کلی: درک کلی خود از دیتاست، مقیاس آن و حوزه کاربرد را بیان کنید.

معرفی ستون‌ها (Features): برای هر یک از ستون‌های موجود، دقیقاً مشخص کنید که چه اطلاعاتی نگهداری می‌کند و نوع داده آن‌ها چیست؟

ارتباطات منطقی: رابطه بین ستون‌ها را بر اساس منطق کسب و کار توضیح دهید.

در پایان این مرحله شما باید درک کاملی از ماهیت هر ستون و نحوه پر شدن داده‌ها در دیتاست داشته باشید

۲-۱- مرحله دوم: طراحی معماری پایه بر اساس Medallion

Bronze Architecture

لایه Bronze یا Raw Layer جایی است که داده‌ها دقیقاً همانطور که از منبع آمده‌اند، بدون هیچگونه تغییر ذخیره می‌شوند.

در دیتابیس PostgreSQL یک Schema bronze جدید به نام bronze ایجاد کنید. یک جدول به نام raw_dataset بسازید.

ستون های جدول را بر اساس تحلیلی که در مرحله اول انجام داده اید و با Data Type مناسب تعریف کنید و دیتاست اصلی را بدون تغییر، در جدول Import کنید.

هدف این مرحله Raw Ingestion است و صرفا حفظ اصالت داده ها مدنظر است.

۲-۲- مرحله سوم: طراحی معماری پایه بر اساس Medallion

Silver Architecture

در این فاز، باید عملیات ETL را انجام دهید تا داده ها برای تحلیل آماده شوند. خروجی این مرحله باید در جدولی با نام silver.cleaned_dataset ذخیره شود.

اقدام اول: مهندسی ویژگی (Feature Engineering)

بررسی کنید آیا فرمت تاریخ و ساعت استاندارد است؟ با نظر خودتان می توانید یک ستون جدید برای استخراج جزئیات زمان یا تاریخ اضافه کنید. دلیل خود را برای این کار بیان کنید. با بررسی کامل دیتاست، تحلیل کنید که آیا با اضافه کردن ستون های جدید، می توان ستون های قدیمی را حذف کرد؟

اقدام دوم: یکپارچه سازی دالیل لغو سفر سه ستون مختلف شامل دالیل لغو توسط راننده، مشتری یا سفر ناقص وجود دارد. یک ستون جدید به نام Unified_cancellation_reason ایجاد کنید. با توجه به ستون Booking Status دلیل لغو صحیح را از یکی از سه ستون برداشته و در ستون جدید ذخیره کنید.

اقدام سوم: مدیریت مقادیر Null بررسی کنید کدام ستون ها حجم زیادی از مقادیر Null دارند. آیا حذف این ردیف ها منطقی است؟ راهکار شما برای مدیریت این حجم Null چیست؟ اقدام چهارم: تحلیل و مدیریت امتیازها ستون های Driver Ratings و Customer Rating را بررسی کنید. Null بودن این مقادیر همیشه به معنی ناقص بودن داده ها نیست. برای وضعیت های مختلف بررسی کنید که Null بودن امتیاز چه معنایی دارد؟ در سفرهای Completed بررسی کنید رفتار مشتریان و رانندگان برای ثبت امتیاز چگونه بوده است؟ آیا با داده های موجود می توانیم برای Customer Rating و سفرهای تکمیل شده چاره ای بیندیشیم تا از حجم Null بودن کم کنیم؟ دقت کنید که از حذف ردیف ها به دلیل نبودن امتیاز خودداری کنید!

۲-۳- مرحله چهارم: طراحی معماری پایه بر اساس Medallion

Gold لایه Architecture

لایه Gold، لایه نهایی و قابل اتکا برای تحلیل کسب و کار و تولید گزارشات است. داده ها در این الیه باید بهینه سازی شده، غنی و آماده خواندن سریع باشند. اقدام اول: طراحی Schema جدول نهایی خود را با نام gold.dataset ایجاد کنید. از انواع داده دقیق و بهینه استفاده کنید. اقدام دوم: محاسبه و غنی سازی داده ها یک ستون جدید محاسباتی اضافه کنید. به عنوان مثال revenue_per_km که حاصل تقسیم داده ها باشد. از محدودیت CHECK برای اطمینان از صحت داده ها استفاده کنید. مثال Primary Key باشد. از محدودیت Primary Key باشد. از محدودیت CHECK برای اطمینان حاصل کنید که امتیازات داده شده بین ۰ تا ۵ است.

۲-۴- مرحله پنجم: پیاده سازی CRUD API

در این مرحله، باید پلی میان کاربر و دیتابیس ایجاد کنید. شما می توانید با هر زبانی یا فریم ورکی که به آن تسلط دارید، مانند Go، Django، Node JS، FastAPI و ... این پیاده سازی را انجام دهید. هدف اصلی این مرحله، انجام عملیات CRUD روی الیه Gold است. اندپوینت های زیر را پیاده سازی کنید: ساخت سفر جدید (Create) با تولید خودکار Booking ID، نمایش لیستی از سفرها (Read) با قابلیت فیلتر بر اساس Customer ID، تغییر وضعیت (Update) و حذف (Delete) یک رکورد سفر از دیتابیس. برای نمایش عملیات CRUD، از Streamlit استفاده کنید.

۲-۵- مرحله ششم: داشبور تحلیلی و تعاملی

در این مرحله با استفاده از Streamlit یا ابزارهای مشابه، یک داشبورد تعاملی بسازید که وضعیت کسب و کار را به تصویر بکشد [۱۳]. اقدام اول: شاخص های کلیدی عملکرد (KPIs) تعداد کل رزروها، تعداد کل رزروهای موفق، مجموع درآمد و نرخ موفقیت سفر [۱۴، ۱۳]. اقدام دوم: نمودارها نمودار Pie Chart برای نمایش درصد یا توزیع دالیل لغو سفر و روش های مختلف پرداخت؛ نمودار Bar Chart برای

مقایسه تعداد سفرهای انجام شده برای هر نوع خودرو و نمایش میانگین امتیاز؛ نمودار خطی برای نمایش ساعت‌های پر تردد و روزهای هفته بر اساس تعداد سفر. اقدام سوم: تعاملی بودن داشبورد باید دارای فیلترهای انتخاب بازه زمانی و انتخاب نوع خودرو باشد. با تغییر فیلترها، تمام شاخص‌ها و نمودارها باید به روز رسانی شوند.

۲-۶- مرحله هفتم: دستیار هوشمند Text to SQL

در این قسمت شما باید یک رابط کاربری چت بسازید که بتواند سوالات مدیران را به زبان انگلیسی دریافت کرده و پاسخ آن را از دیتابیس استخراج کند. معماری Workflow: ورودی سوال متني کاربر به زبان انگلیسی است. تولید SQL توسط AI با دادن اطلاعات ساختار جدول Gold به مدل زبانی انجام می‌شود. اعتبارسنجی و فیلتر: کد شما باید خروجی مدل را قبل از اجرا بررسی کند؛ دستورات مخرب مانند DELETE، DROP، UPDATE و INSERT باید بالک شوند. الگوهای خطرناک مانند SELECT * بدون LIMIT باید بالک شوند (اگر کوئری بدون LIMIT بود، LIMIT 10 را به آن اضافه کنید). سیستم شما فقط باید به درخواست‌های SELECT پاسخ دهد.

۲-۷- مرحله هشتم: بهینه سازی و ایندکس گذاری

در این مرحله باید به صورت عملی تاثیر ایندکس گذاری بر بهبود عملکرد دیتابیس را بسنجید. اقدام اول: انتخاب و نوشتن کوئری سنگین یک کوئری SELECT بنویسید که شامل فیلترهای ترکیبی یا عملیات Aggregation باشد. اقدام دوم: قبل از ایجاد ایندکس کوئری را روی جدول Gold اجرا کنید و از دستور EXPLAIN ANALYZE استفاده کنید تا زمان اجرای واقعی و پلن اجرایی را مشاهده کنید. اقدام سوم: ایندکس گذاری با توجه به شناخت از دیتاست، یک یا چند ایندکس مناسب (مانند Composite Index) بسازید. دلیل انتخاب ستون‌های ایندکس را به صورت تکنیکال توجیه کنید. مجدداً همان کوئری دقیق را اجرا کرده و زمان اجرای آن را قبل و بعد از ایندکس گذاری مقایسه کنید.

۲-۸ - مرحله نهم: پیاده سازی Vector Database و جستجوی معنایی (Semantic Search)

در این مرحله، شما باید یک دیتابیس برداری (Vector Database) را به پروژه اضافه کنید تا قابلیت جستجوی معنایی روی دالیل لغو سفر ایجاد کنید. اقدام اول: انتخاب ابزار و طراحی دیتابیس برداری از Booking ID استفاده کنید. ستون Unified_cancellation_reason را به همراه Chroma DB در Chroma DB Collection در DB وارد کنید. اقدام دوم: پیاده سازی اندپوینت جستجو یک بخش جستجو جدید بسازید که یک عبارت متنی درباره علت کنسل شدن سفرها از کاربر دریافت کند. این عبارت را در Chroma DB جستجو کنید تا ۵ مورد ($Top_k = 5$) که بیشترین شباهت معنایی را دارند برگرداند. سیستم باید از شناسه های سفر (Booking ID) برای واکشی جزئیات کامل از جدول gold.dataset در PostgreSQL استفاده کرده و نتیجه را به کاربر نمایش دهد.

پیاده‌سازی پروژه

گزارش فاز اول: تحلیل و شناسایی مجموعه داده (Data Analysis)

۱-۳-۲- توصیف کلی (General Description)

خب اول از همه فایل رو با Excel باز کردیم، چند سطر اول و آخر رو نگاه کردیم و بعد با Python و کتابخانه pandas یک خلاصه سریع گرفتیم تا متوجه بشیم دقیقاً چی و چه دیتاهایی داخل این دیتاست هست.

این دیتاست شامل ۱۴۸,۷۷۰ رکورد از تراکنش‌های واقعی شرکت Uber در سال ۲۰۲۴ است.

ما با یک سیستم کاملاً زنده و عملیاتی طرف هستیم. این دیتاست صرفاً چند تا عدد و رقم ساده نیست این داده‌ها همه چیزهایی رو که تو یک سفر Uber اتفاق می‌افته ثبت کردن: از لحظه‌ای که مشتری درخواست خودش رو داده، تا وقتی سفر تمام می‌شه یا لغو می‌شه. به همین دلیل هم مقیاس دیتا واقعاً بزرگه (نزدیک ۱۵۰ هزار سفر)، پس کاملاً مناسب تحلیل‌های واقعی کسب‌وکار و عملیات‌های تحلیل و آنالیز داده هست.

حوزه کاربردش هم بسیار واضح هست: کمک به مدیر شرکت Uber که بفهمه کدوم ساعات شلوغ تر هست، مردم بیشتر با چه ماشین‌هایی سفر می‌کنند؟ چرا سفرها لغو می‌شوند؟ درآمد از کجا بدست آمده و...

کاربرد اصلی این داده‌ها برای ما در اینجا، این هست که بفهمیم کجاها سیستم داره ضرر میده مثلًا چرا سفرها لغو می‌شن؟ چون نرخ لغو سفر بالا هست! و کجاها بیشترین درآمد رو داره؟، عملکرد رانندگان و مشتری‌ها چطوری بوده؟ تا بتونیم الگوهای پنهان داده‌ها و سفرها رو بیرون بکشیم و در نهایت به بهینه‌سازی درآمد بر اساس نوع وسیله نقلیه (از موتور و دوچرخه برقی تا ماشین‌های لوکس و...) و مسافت و بقیه پارامترها بتونیم بررسیم.

۲-۳-۲ - تحلیل ستون‌ها و روابط بین آن‌ها

حالا برایم سراغ ستون‌ها. من هر کدوم رو نگاه کردم و سعی کردم بفهمم چی نگه می‌دارم و از چه نوعی تشکیل شده‌اند. مثلا:

- Date: تاریخ سفر که قطعاً باید از نوع DATE باشد چون تاریخ کامل مورد نیاز است.
- Time: زمان دقیق، مثل 12:29:38 که باید از نوع TIME یا TIMESTAMP باشد. این ستون البته می‌توانه با تاریخ ترکیب بشود در آینده.
- Booking ID: شناسه منحصر به فرد هر رزرو مثل CNR5884300، که می‌توانه از نوع VARCHAR تعریف بشود و قاعده‌تا می‌توانه کلید اصلی باشد.
- Booking Status: وضعیت سفر، مانند Completed یا Incomplete که باز هم VARCHAR مناسب است.
- Customer ID: شناسه مشتری، مثل بالا از نوع VARCHAR.
- Vehicle Type: نوع وسیله، مثل eBike یا Auto، از نوع VARCHAR.
- Cancelled Rides by Customer: لغو سفر توسط مشتری که یک فلگ است و null یا INTEGER (عدد ۱) است. این می‌توانه بعداً به بولین تبدیل شود.
- Reason for cancelling by Customer: دلیل لغو توسط مشتری، که TEXT مناسب است.
- Cancelled Rides by Driver: لغو سفر توسط راننده که یک فلگ است و null یا INTEGER (عدد ۱) است. این می‌توانه بعداً به بولین تبدیل شود.
- Driver Cancellation Reason: دلیل لغو سفر توسط راننده، که TEXT است.
- Incomplete Rides: تعداد سفرهای تکمیل نشده، که عدد ساده یا INTEGER است.
- Incomplete Rides Reason: دلیل سفر تکمیل نشده TEXT است.
- Booking Value: ارزش مالی که برای پول DECIMAL است.
- Ride Distance: مسافت DECIMAL است.

▪ امتیاز راننده که عددی بین ۰ تا ۵ است و DECIMAL هست.

▪ امتیاز مشتری، مشابه قبلی Customer Rating.

▪ روش پرداخت که Payment Method VARCHAR.

این ستون‌ها پر از null هستند که منطقیه چون مثلاً اگر سفر کامل بشه، دلایل لغو خالی می‌مونه.

در مورد ارتباطات، همانطور که مشخص هست، همه چیز رو یک رزرو با Booking ID تعریف می‌کنه. برای مثال Customer ID به رزروها وصل می‌شه، یعنی یک مشتری می‌تونه چند سفر داشته باشه و هر سفر مربوط به یک مشتری هست.

Booking Status تعیین می‌کنه کدوم دلایل لغو سفر پر بشوند اگر Completed باشه، بقیه null هستن.

Vehicle Type و Payment Method هم به رزرو وابسته‌اند و می‌تونن برای تحلیل گروهی استفاده بشوند. مسافت و Booking Value هم با هم مرتبط هستند و می‌شه ازشون درآمد حساب کرد. امتیازها فقط برای سفرهای موفق معنی دارند و null بودنشون، نشون‌دهنده این هست که طرف امتیازی نداده است. کلا، داده‌ها طوری هستن که می‌تونیم الگوهای زمانی (از Date و Time) یا رفتاری (مثلاً از لغوها) رو استخراج کنیم.

برای اینکه بهتر نشون بدیم، یک دیاگرام ER ساده کشیدم که موجودیت‌ها و روابط رو نشون می‌ده. این صرفاً بر اساس فهم خودم از دیتابست هست:

اگر وضعیت سفر "No Driver Found" باشه، عملاً ستون‌های قیمت و مسافت و امتیاز همگی خالی (null) می‌مونن چون اصلاً سفری شروع نشده که بخواهد قیمتی داشته باشه.

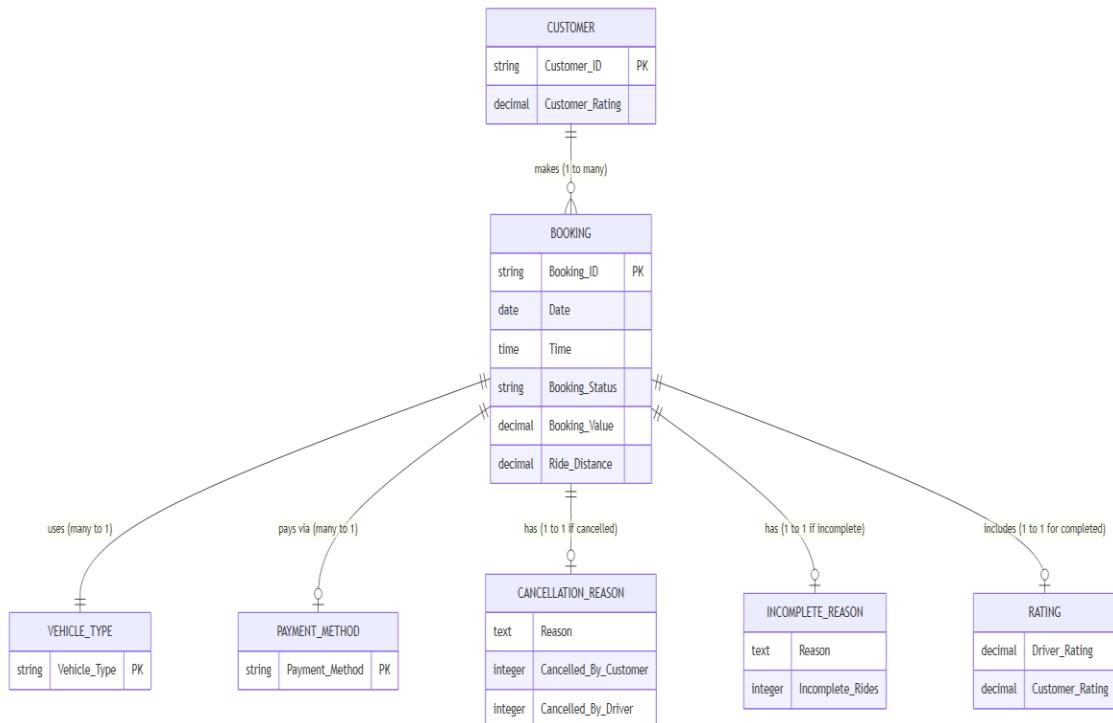
یک رابطه مستقیم بین "Vehicle Type" و "Booking Value" وجود داره؛ مثلاً قطعاً قیمت یک ماشین لوکس در یک مسیر مشخص باید از یک وسیله نقلیه ارزان بیشتر باشه.

در این بخش، ماهیت هر ستون و نوع داده‌ای پیشنهادی برای ذخیره‌سازی بهینه در PostgreSQL در معرفی شده است:

جدول ۱ معرفی ستون ها و نوع داده ها (Featuring & Data Types)

نام ستون در دیتابست	شرح اطلاعات	نوع داده (Data Type)
Date	تاریخ ثبت درخواست سفر	DATE
Time	زمان دقیق ثبت درخواست	TIME
Booking ID	(Primary Key) شناسه منحصر به فرد برای هر تراکنش	VARCHAR(20)
Booking Status	(Completed, Cancelled, etc.) وضعیت فعلی سفر	VARCHAR(50)
Customer ID	شناسه منحصر به فرد مشتری	VARCHAR(20)
Vehicle Type	(eBike, Go Sedan, Auto, etc.) نوع وسیله نقلیه	VARCHAR(30)
Cancelled Rides by Customer	نشانگر لغو توسط مشتری (معمولًا null یا ۱)	BOOLEAN یا SMALLINT
Reason for cancelling by Customer	علت لغو سفر از سمت مشتری	TEXT
Cancelled Rides by Driver	نشانگر لغو توسط راننده (معمولًا null یا ۱)	BOOLEAN یا SMALLINT
Driver Cancellation Reason	علت لغو سفر از سمت راننده	TEXT
Incomplete Rides	نشانگر سفرهایی که نیمه کاره مانده‌اند	BOOLEAN یا SMALLINT
Incomplete Rides Reason	علت اتمام نیمه کاره سفر (مانند نقص فنی)	TEXT
Booking Value	مبلغ کل تراکنش	DECIMAL(10, 2)
Ride Distance	مسافت طی شده (بر حسب واحد مسافت)	DECIMAL(8, 2)
Driver Ratings	امتیاز داده شده به راننده (۰ تا ۵)	DECIMAL(2, 1)
Customer Rating	امتیاز داده شده به مشتری (۰ تا ۵)	DECIMAL(2, 1)
Payment Method	(Cash, UPI, Credit Card, etc.) روش پرداخت	VARCHAR(20)

حالا برای اینکه روابط بین موجودیت ها را بهتر نشون بدم، یک دیاگرام ER ساده کشیدم که موجودیت ها و روابط رو نشون می ده. این بر اساس فهم خودم از دیتابست هست:



شکل ۱ دیاگرام موجودیت ها و روابط (ER Diagram)

گزارش فاز دوم: Bronze Medalion Architecture

این لایه Raw Layer هست یا داده خام و هدفش ذخیره داده‌ها بدون هیچ تغییری روی اون هاست. داده‌ها رو مستقیم از CSV ایمپورت می‌کنیم و وارد دیتابیس انتخابی که من از PostgreSQL استفاده کردم که یکی از بهترین و محبوب‌ترین دیتابیس‌های Relational هست.
من تمامی مراحل رو کامل در این گزارش می‌نویسم:

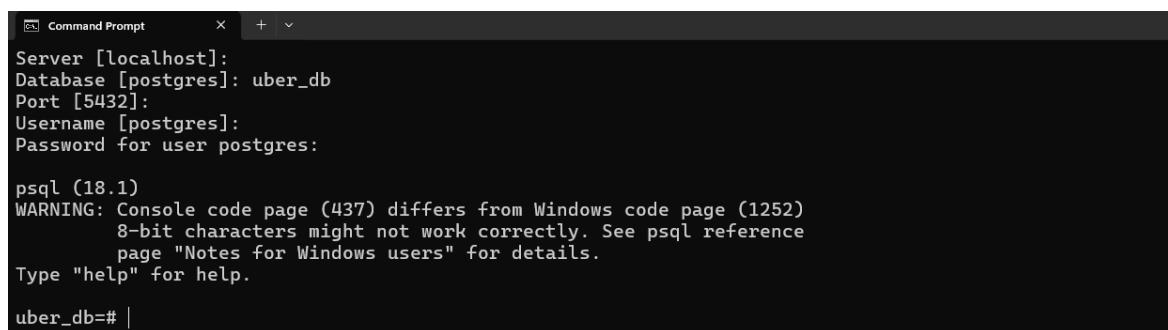
۱. دانلود PostgreSQL از این آدرس که وبسایت رسمی پیشنهاد داده:

<https://www.enterprisedb.com/downloads/postgres-postgresql-downloads/>

این نسخه لوکال روی ماشین ما نصب می‌شود. می‌توانستیم با استفاده از داکر یا روش‌های دیگر هم نصب کنیم ولی من ترجیح دادم روی سیستم و به صورت لوکال نصب کنم.

۲. ساخت جدول در پایگاه داده:

پس از نصب، از Shell psql استفاده می‌کنیم و راهاندازی اولیه رو انجام می‌دهیم:



```
PS C:\> Command Prompt
Server [localhost]: Database [postgres]: uber_db
Port [5432]:
Username [postgres]:
Password for user postgres:

psql (18.1)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

uber_db=#
```

شکل ۲ راهاندازی اولیه PostgreSQL از طریق Shell psql

حال با دستور زیر جدول اولمان رو ایجاد می‌کنیم:

```
CREATE DATABASE uber_db;
```

۳. حال کافی است اسکیمای اولیه برنز و جدول رو بر اساس نوع داده‌هایی که در قسمت اول و فاز اول تعریف و تحلیل کردیم، تعریف کنیم:

```
CREATE TABLE bronze.raw_dataset (
    date DATE,
    time TIME,
    booking_id VARCHAR(20),
    booking_status VARCHAR(50),
    customer_id VARCHAR(20),
    vehicle_type VARCHAR(50),
    cancelled_by_customer INTEGER,
    customer_cancel_reason TEXT,
    cancelled_by_driver INTEGER,
    driver_cancel_reason TEXT,
    incomplete_rides INTEGER,
    incomplete_reason TEXT,
    booking_value DECIMAL(10,2),
    ride_distance DECIMAL(10,2),
    driver_rating DECIMAL(2,1),
    customer_rating DECIMAL(2,1),
    payment_method VARCHAR(50)
);
```

این کد صرفا جدول داده‌های Bronze از معماری مدل‌الیون رو با توجه به شناخت و تحلیلی که در فاز و قسمت اول انجام دادیم ایجاد می‌کند. که این کد در محیط PostgreSQL باید در psql اجرا شود.

```
uber_db=# CREATE SCHEMA IF NOT EXISTS bronze;
CREATE SCHEMA
uber_db=#
uber_db=# CREATE TABLE IF NOT EXISTS bronze.raw_dataset (
uber_db(#     date DATE,
uber_db(#     time TIME,
uber_db(#     booking_id VARCHAR(20),
uber_db(#     booking_status VARCHAR(50),
uber_db(#     customer_id VARCHAR(20),
uber_db(#     vehicle_type VARCHAR(50),
uber_db(#     cancelled_by_customer INTEGER,
uber_db(#     customer_cancel_reason TEXT,
uber_db(#     cancelled_by_driver INTEGER,
uber_db(#     driver_cancel_reason TEXT,
uber_db(#     incomplete_rides INTEGER,
uber_db(#     incomplete_reason TEXT,
uber_db(#     booking_value DECIMAL(10,2),
uber_db(#     ride_distance DECIMAL(10,2),
uber_db(#     driver_rating DECIMAL(2,1),
uber_db(#     customer_rating DECIMAL(2,1),
uber_db(#     payment_method VARCHAR(50)
uber_db(# );
CREATE TABLE
uber_db=# |
```

شکل ۳ ساختن اسکیمای Bronze و ساخت جدول لایه اول Bronze Layer

۴. کافی است داده‌های خام را از فایل اولیه که من به original_dataset تغییر نام دادم و داخل پوشه database در پروژه ام قرار دادم ایمپورت کنیم و سپس بدون هیچ تغییری آن را داخل دیتابیس و جدولی که ایجاد کردیم، ذخیره کنیم. برای ایمپورت کردن داده‌ها از فایل CSV، ما یک اسکریپت Node.js می‌نویسیم که در انجام فرایند ETL به ما کمک می‌کند و بخشی از کار اولیه ماست.

نکته حائز اهمیت که خیلی خیلی هم مهم هست اینه که برای درج اطلاعات از فایل CSV به خود دیتابیس و جداول PostgreSQL، مقادیری مثل "null" رشته‌ای به عنوان string داخل CSV وجود دارد که این‌ها رو PostgreSQL نمی‌تونه به عدد یا نوع NULL واقعی تبدیل کنه و این ارور رو قاعده‌تا خواهیم گرفت:

```
error: invalid input syntax for type integer: "null"
      at D:\Daneshgoh\Master\Term 1\Advance Data Base\T
      amrin\Project\uber_data_platform\backend\node_modules
      \pg\lib\client.js:624:17
          at process.processTicksAndRejections (node:internal/
          process/task_queues:103:5)
```

شکل ۴ ارور وجود null رشته‌ای در فایل .csv

برای همین باید توجه کنیم که کد نیاز به اصلاح داره تا این string‌ها رو به null تبدیل کنه قبل از INSERT کردن.

برای همین این مورد رو به صورت دستی هندل می‌کنیم:

```
const values = Object.values(row).map((value) =>
    value === "null" ? null : value
);
```

در نهایت کد نهایی که داده‌ها رو از فایل اکسل csv. می‌گیره و داخل Database ایمپورت می‌کنه به صورت زیر نوشته شده که در پایان هم پیغام Import done با موفقیت چاپ می‌شه و این به معنای اتمام لایه برنز Bronze است. چون در لایه Bronze (طبق Medallion Architecture)، هدف Raw Ingestion هست.

¹: به معنی استخراج - انتقال داده - بارگزاری داده است Extract Transform Load

```

const fs = require("fs");
const csv = require("csv-parser");
const pool = require("./models/db.js");

const importData = async () => {
  const client = await pool.connect();
  fs.createReadStream("../database/original_dataset.csv")
    .pipe(csv())
    .on("data", async (row) => {
      const values = Object.values(row).map((value) =>
        value === "null" ? null : value
      );
      await client.query(
        `INSERT INTO bronze.raw_dataset VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13, $14, $15, $16, $17)`,
        values
      );
    })
    .on("end", () => console.log("Import done"));
};

importData();

```

شکل ۵ کد وارد کردن داده‌های خام از فایل csv به داخل raw_dataset و جدول bronze schema

همچنین اتصال به دیتابیس هم توسط این کد خیلی ساده انجام می‌شود و اتصال به پایگاهداده برقراره:

```

const { Pool } = require('pg');
const pool = new Pool({
  user: 'postgres',
  host: 'localhost',
  database: 'uber_db',
  password: 'Admin@123456',
  port: 5432,
});

module.exports = pool;

```

Connection Information	
Parameter	Value
Database	uber_db
Client User	postgres
Host	localhost
Host Address	::1
Server Port	5432
Options	
Protocol Version	3.0
Password Used	true
GSSAPI Authenticated	false
Backend PID	11228
SSL Connection	false
Superuser	on
Hot Standby	off

(13 rows)

این کد خیلی ساده بکاند رو به دیتابیس متصل می‌کند.

شکل ۶ چک کردن اتصال به PostgreSQL

خب در ادامه تمامی جداول و اسکیماهایی که داخل PostgreSQL ساخته شده‌اند رو نمایش می‌دهیم:

List of tables			
Schema	Name	Type	Owner
bronze	raw_dataset	table	postgres
information_schema	sql_features	table	postgres
information_schema	sql_implementation_info	table	postgres
information_schema	sql_parts	table	postgres
information_schema	sql_sizing	table	postgres
pg_catalog	pg_aggregate	table	postgres

شکل ۷ اسکیماها و جداول ساخته شده در PostgreSQL DB

در نهایت این هم جدول ما داخل PostgreSQL است:

Table "bronze.raw_dataset"					
Column	Type	Collation	Nullable	Default	
date	date				
time	time without time zone				
booking_id	character varying(20)				
booking_status	character varying(50)				
customer_id	character varying(20)				
vehicle_type	character varying(50)				
cancelled_by_customer	integer				
customer_cancel_reason	text				
cancelled_by_driver	integer				
driver_cancel_reason	text				
incomplete_rides	integer				
incomplete_reason	text				
booking_value	numeric(10,2)				
ride_distance	numeric(10,2)				
driver_rating	numeric(2,1)				
customer_rating	numeric(2,1)				
payment_method	character varying(50)				

شکل ۸ لایه Bronze در raw_dataset

گزارش فاز سوم: Silver Medalion Architecture لایه

خب برای انجام این قسمت اول از همه، schema و جدول silver_cleaned_dataset را می‌سازیم. این جدول همون داده‌های تمیز شده و غنی‌شده‌ست که برای تحلیل و لایه Gold قسمت بعد آماده می‌شوند.

با توجه به دیتاست و اهداف مرحله نقره‌ای(Silver)، تمرکز اصلی بر تمیزسازی، یکپارچه‌سازی و مهندسی ویژگی است تا داده‌ها برای تحلیل آماده شوند. در این مرحله نباید رکوردها حذف شوند مگر در موارد بسیار ضروری، زیرا مقادیر NULL در این دیتاست اغلب معنای تجاری دارند و نبود یک رویداد را نشان می‌دهند. من تمام عملیات را در یک تراکنش نوشته‌ام تا اگر خطای رخ دهد، همه چیز به حالت قبل برگردد. در ادامه هر یک از چهار اقدامی که در صورت پروژه برای این مرحله مشخص شده را به‌همراه کد مربوطه و توضیحات لازم می‌آورم و به آن پرداخته خواهد شد.

۲-۹- راهاندازی اولیه تراکنش

```
require('dotenv').config();
const pool = require('../models/db.js');
async function runSilverETL() {
  const client = await pool.connect();
  try {
    console.log('⌚ Starting Silver Layer ETL ...');
    await client.query('BEGIN');
    // ... Implementation
    await client.query('COMMIT');
    console.log('⌚ Silver Layer ETL completed successfully!');
  } catch (err) {
    await client.query('ROLLBACK');
    console.error('✖ Silver Layer ETL failed:', err);
    throw err;
  } finally {
    client.release();
  }
}
runSilverETL().catch(console.error);
```

خب اول از همه کتابخانه‌های لازم رو استفاده کردیم dotenv رو برای خوندن متغیرهای محیطی (مثل اطلاعات دیتابیس) استفاده می‌کنیم. بعدش با pool از فایل db.js یه کانکشن از دیتابیس می‌گیریم. کل عملیات رو توی یه try/catch گذاشتیم که اگه خطای پیش بیاد، تراکنش رو ROLLBACK و اگه همه چیز طبق خواسته ما پیشبره، در انتهای COMMIT می‌کنیم. این کار باعث میشه داده‌ها همیشه در یک وضعیت consistent و سازگار باشند. لاگ‌ها رو هم با پیغام مناسب ایجاد کردیم.

۲-۱- اقدام اول مهندسی ویژگی (Feature Engineering)

اصلاً مهندسی ویژگی یعنی چه؟ فرآیند تبدیل داده‌های خام به متغیرها (ویژگی‌هایی) است که بهتر بتوانند ماهیت بیزینس را به نمایش بگذارند. در واقع ما داده‌ها را دستکاری می‌کنیم تا هوشمندتر شوند و در مراحل بعدی (مثل تحلیل در داشبورد یا پرسش‌وپاسخ هوشمند) نتایج دقیق‌تری بگیریم.

```
-- Create silver schema and cleaned_dataset table with feature-engineered columns
CREATE SCHEMA IF NOT EXISTS silver;
CREATE TABLE IF NOT EXISTS silver.cleaned_dataset (
    booking_id          TEXT PRIMARY KEY,
    customer_id         TEXT,
    vehicle_type        TEXT,
    booking_status      TEXT,
    booking_value       NUMERIC(10,2),
    ride_distance       NUMERIC(10,2),
    payment_method      TEXT,
    trip_timestamp      TIMESTAMP,
    pickup_hour         INTEGER,
    day_of_week         INTEGER,    -- 0=Sunday, 6=Saturday
    day_name             TEXT,      -- Monday, Tuesday, ...
    month                INTEGER,
    year                 INTEGER,
    is_weekend           BOOLEAN,
    unified_cancellation_reason TEXT,
    driver_rating        NUMERIC(3,2),
    customer_rating      NUMERIC(3,2),
    driver_rating_imputed BOOLEAN DEFAULT FALSE,
    customer_rating_imputed BOOLEAN DEFAULT FALSE,
    is_cancelled         BOOLEAN
);
```

ابتدا اسکیمای silver را ساختیم. سپس جدول اصلی یعنی cleaned_dataset را ایجاد کردہ‌ایم. ستون‌هایی که از Bronze می‌آیند را با DataType مناسب تعریف کردہ‌ایم. چند ستون جدید هم برای مهندسی ویژگی اضافه کردیم که در زیر توضیح می‌دهیم:

اولین چیزی که توی دیتاست برنز به چشم می‌خورد، جدا بودن ستون‌های Date و Time با فرمت غیراستاندارد است. برای اینکه بتونیم تحلیل‌های زمانی راحت‌تری انجام بدهیم. باید این دو تا ستون رو باهم ترکیب کنیم و به یه تایم‌استمپ واقعی تبدیل کنیم.

که ترکیب تاریخ و زمان هست، طبق اشاره صورت پروژه و از اون pickup_hour برای ساعت سفر و روز هفته day_of_week به صورت عددی برای مثال ۰ = Saturday، نام روز day_name (مانند: Monday) و month و year استخراج می‌شوند. هم که نشون میده سفر آخر هفته بوده یا نه.

اینا همه کمک می‌کنند تا در مراحل بعدی دیگه نیازی به دستکاری تاریخ و زمان نداشته باشیم. دو ستون customer_rating_imputed و driver_rating_imputed نیز با مقدار پیش‌فرض FALSE گذاشتیم تا بعداً وقتی امتیازها رو جایگزین کردیم، پرچم‌داشته باشند و مشخص باشه کدوم مقادیر رو ما خودمون جایگزین کرده‌بودیم. ستون is_cancelled رو هم به صورت بولین اضافه کردم تا خیلی راحت بتونم سفرهای لغو شده رو فیلتر کنم.

۱-۳-۲ - پاکسازی داده‌های قبلی و درج اولیه از برنز

```
TRUNCATE silver.cleaned_dataset;
```

قبل از اینکه داده‌های جدید رو وارد کنم، با TRUNCATE جدول نقره‌ای رو خالی کردم تا مطمئن بشم داده‌های تکراری و قدیمی نداشته باشیم.

```

INSERT INTO silver.cleaned_dataset (
    booking_id, customer_id, vehicle_type, booking_status,
    booking_value, ride_distance, payment_method,
    trip_timestamp,
    pickup_hour, day_of_week, day_name, month, year, is_weekend,
    unified_cancellation_reason,
    driver_rating, customer_rating,
    is_cancelled
)
SELECT
    "Booking ID",
    "Customer ID",
    "Vehicle Type",
    "Booking Status",
    NULLIF("Booking Value", '')::NUMERIC,
    NULLIF("Ride Distance", '')::NUMERIC,
    "Payment Method",
    TO_TIMESTAMP("Date" || ' ' || "Time", 'MM/DD/YYYY HH24:MI:SS'),
    EXTRACT(HOUR FROM TO_TIMESTAMP("Date" || ' ' || "Time", 'MM/DD/YYYY HH24:MI:SS')),
    EXTRACT(DOW FROM TO_TIMESTAMP("Date" || ' ' || "Time", 'MM/DD/YYYY HH24:MI:SS')),
    TO_CHAR(TO_TIMESTAMP("Date" || ' ' || "Time", 'MM/DD/YYYY HH24:MI:SS'), 'FMDay'),
    EXTRACT(MONTH FROM TO_TIMESTAMP("Date" || ' ' || "Time", 'MM/DD/YYYY HH24:MI:SS')),
    EXTRACT(YEAR FROM TO_TIMESTAMP("Date" || ' ' || "Time", 'MM/DD/YYYY HH24:MI:SS')),
    CASE WHEN EXTRACT(DOW FROM TO_TIMESTAMP("Date" || ' ' || "Time", 'MM/DD/YYYY HH24:MI:SS')) IN (0, 6)
        THEN TRUE ELSE FALSE END,
    ...
)

```

برای درج دستور INSERT رو نوشتم. اینجا از NULLIF استفاده کردم تا رشته‌های خالی که ممکنه توی ستون‌های عددی باشن به NULL تبدیل بشن و موقع تبدیل به عدد خطانده‌ند سپس با TO_TIMESTAMP و مشخص کردن دقیق فرمت ورودی، تاریخ و زمان رو به یه تایم‌استمپ استاندارد تبدیل کردم. همون جا و بلافاصله ویژگی‌های is_weekend, pickup_hour, day_of_week, day_name, month, year رو استخراج کردم.

۲-۲- اقدام دوم: یکپارچه‌سازی دلایل لغو سفر

سه ستون جداگانه برای دلیل لغو داشتیم که بسته به وضعیت سفر پر می‌شدند. من یک ستون unified_cancellation_reason با توجه به صورت پروژه ساختم که همه دلایل رو در یک جا جمع می‌کنه بدون نیاز به خواندن چندین ستون. من به جای اینکه به booking_status تکیه کنم و با ILIKE رشته‌ها رو مقایسه کنم (که ممکنه توی داده‌های واقعی جواب نده)، تصمیم گرفتم مستقیم برم سراغ خود ستون‌های نشانگر لغو. یعنی اگه ستون Cancelled Rides by Customer غیرتهی بود، یعنی دلیل لغو مشتری توی ستون بعدش وجود داره. همین منطق رو برای راننده و سفر ناتمام هم به کار بردم. به این ترتیب یه ستون واحد unified_cancellation_reason ساخته شد:

```
-- -- Unified cancellation reason: pick the reason from the column that has a non-null indicator
CASE
    WHEN "Cancelled Rides by Customer" IS NOT NULL THEN "Reason for cancelling by Customer"
    WHEN "Cancelled Rides by Driver"     IS NOT NULL THEN "Driver Cancellation Reason"
    WHEN "Incomplete Rides"           IS NOT NULL THEN "Incomplete Rides Reason"
    ELSE "Reason Not Specified"
END,
```

نکته حائز اهمیت این است که برای جلوگیری از افزایش تعداد null ها در ستون جدید ما اگه دلیل کنسل شدن سفر رو نداشته باشیم: null نمی‌گذاریم! بلکه با عبارت "Reason Not Specified" اون رو پر می‌کنیم. این کار معمولا در Data Warehouse خیلی بهتر است. این کار هم جدول رو جمع‌وجورتر کرده و هم کوئری‌زدن برای تحلیل دلایل لغو رو فوق‌العاده ساده‌تر می‌کنه.

این کار همچنین یکی از ایده‌های مدیریت مقادیر null هم هست که در قسمت بعدی خواسته شده بود.

۲-۳- اقدام سوم: مدیریت مقادیر Null

وقتی داده‌های بزرگ رو بررسی کردم، دیدم که ستون‌های booking_value، incomplete_rides، cancelled_by_driver، cancelled_by_customer مثل ride_distance کلی مقدار NULL دارند. مهم‌تر از همه، بعضی از رشته‌های خالی هم وجود داشت که

موقع تبدیل به عدد باعث خطا می‌شد. اولین کاری که کردم این بود که با `NULIF` مطمئن شدم رشته‌های خالی واقعاً به `NUL` تبدیل بشن، نه اینکه خطا بخواهیم بدیم!

اما نکته اصلی اینه که من حتی یک ردیف رو هم حذف نکرم. چرا؟ چون توی این دیتاست، `NUL` بودن خیلی از ستون‌ها معنای تجاری داره. مثلاً وقتی سفر کنسل شده، طبیعی است که مسافت یا مبلغ سفر ثبت نشده باشه.

```
-- Convert empty strings to NULL for numeric fields
NULIF("Booking Value", '')::NUMERIC AS booking_value,
NULIF("Ride Distance", '')::NUMERIC AS ride_distance,
```

و تنها جایی که واقعاً باید فکری به حال `NUL` ها می‌کرم، ستون‌های امتیاز بود که اونم توی اقدام چهارم جداگانه حلش کردم. خلاصه اینکه استراتژی من در مدیریت `NUL` این بود که: `NUL` های منطقی رو حفظ کن، فقط اونایی رو که میشه با یه مقدار معقول پر کرد (مثل امتیاز) رو جایگزین کن، و هرگز ردیف رو پاک نکن.

۴-۲-۴ - اقدام چهارم: تحلیل و مدیریت امتیازها

بزرگترین چالش این مرحله، ستون‌های `customer_rating` و `driver_rating` بودند. در سفرهای کامل شده، تعداد زیادی `NUL` دیدم که نشون می‌داد راننده یا مشتری بعد از اصلاً نخواسته امتیاز بده و نه اینکه داده‌ها مشکل داشته باشند همچنین: در صورت پروژه صراحتاً گفته بود «از حذف ردیف‌ها به دلیل نبودن امتیاز خودداری کنید». پس راهی که به نظر می‌رسه پر کردن سلول‌ها با مقادیر جدید هست.

اول فکر کردم یک میانگین کلی بگیرم، ولی بعد گفتم این کار عادلانه نیست؛ چون ممکنه خودروهای لوکس امتیاز بالاتری داشته باشن و خودروهای اقتصادی پایین‌تر و اگه که میانگین کلی رو جایگزین کنم، تحلیل‌های بعدی رو به هم می‌ریزم. برای همین رفتم سراغ محاسبه میانگین به تفکیک نوع خودرو (`vehicle_type`). در کوئری زیر با `WITH` میانگین هر خودرو رو حساب کردم و بعد توی `UPDATE`، اگه امتیاز `NUL` بود، مقدار میانگین همون نوع خودرو رو گذاشتم. حالا اگر هم

بر حسب اتفاق میانگین هم NULL می‌شد (یعنی هیچ امتیازی برای اون خودرو ثبت نشده بود)، از به مقدار پیش‌فرض ۴.۵ استفاده کردم. که در داده‌های uber معمولاً میانگین همین تقریباً نزدیکه به همین

عدد است این منبع [تایید میکنید](https://www.uber.com/en-US/newsroom/riderratingsranking2024/)

دو تا ستون بولین هم به اسم customer_rating_imputed و driver_rating_imputed اضافه کردم تا مشخص بشه کدام امتیازها واقعی‌اند و کدام‌ها جایگزین شدن. این کار هم مشکل NULL را حل کرد، هم دقت رو بالاتر برد. کدهای این بخش رو هم آوردم:

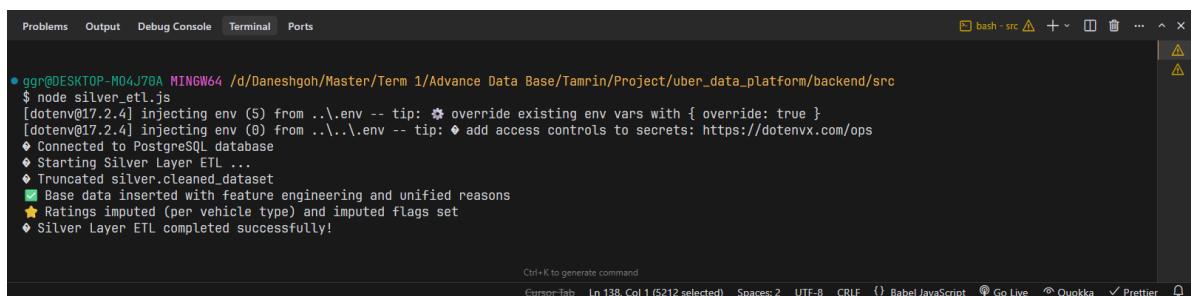
```
-- Calculate average ratings per vehicle type for completed trips only
WITH avg_per_vehicle AS (
    SELECT
        vehicle_type,
        AVG(driver_rating) FILTER (
            WHERE booking_status = 'Completed' AND driver_rating IS NOT NULL
        ) AS avg_driver,
        AVG(customer_rating) FILTER (
            WHERE booking_status = 'Completed' AND customer_rating IS NOT NULL
        ) AS avg_customer
    FROM silver.cleaned_dataset
    GROUP BY vehicle_type
)
-- Update null ratings with vehicle-specific averages and set imputed flags
UPDATE silver.cleaned_dataset s
SET
    driver_rating = COALESCE(s.driver_rating, a.avg_driver, 4.5),
    customer_rating = COALESCE(s.customer_rating, a.avg_customer, 4.5),
    driver_rating_imputed = CASE
        WHEN s.driver_rating IS NULL AND s.booking_status = 'Completed' THEN TRUE
        ELSE FALSE
    END,
    customer_rating_imputed = CASE
        WHEN s.customer_rating IS NULL AND s.booking_status = 'Completed' THEN TRUE
        ELSE FALSE
    END
FROM avg_per_vehicle a
WHERE s.vehicle_type = a.vehicle_type
```

```
AND s.booking_status = 'Completed';
```

۲-۵ - پایان تراکنش و لاغ نهایی

اگر که همه‌ی این مراحل با موفقیت انجام بشن، COMMIT می‌شود تراکنش و پیام موفقیت چاپ می‌شود. در غیر این صورت ROLLBACK و خطا نشون داده می‌شود و سیستم به وضعیت قبلی خود به طور کامل برمی‌گردد.

کد کامل که تمامی اقدامات رو در برگرفته رو اجرا می‌کنیم و مرحله به مرحله گزارش می‌دهیم:



```
gqr@DESKTOP-M04J70A MINGW64 /d/Daneshgoh/Master/Term 1/Advance Data Base/Tamrin/Project/uber_data_platform/backend/src
$ node silver_etl.js
[dotenv@17.2.4] injecting env (5) from ..\env -- tip: * override existing env vars with { override: true }
[dotenv@17.2.4] injecting env (6) from ..\..\env -- tip: ♦ add access controls to secrets: https://dotenvx.com/ops
◆ Connected to PostgreSQL database
◆ Starting Silver Layer ETL ...
◆ Truncated silver_cleaned_dataset
✓ Base data inserted with feature engineering and unified reasons
★ Ratings imputed (per vehicle type) and imputed flags set
♦ Silver Layer ETL completed successfully!
```

شکل ۹ اجرای موفق کد silver_etl و لایه silver

برای مقایسه بهتر و مشاهده تفاوت‌ها ابتدا ساختار جدول Bronze رو نشان می‌دهیم:

bronze.raw_dataset					
Column	Type	Collation	Nullable	Default	
date	date				
time	time without time zone				
booking_id	character varying(20)				
booking_status	character varying(50)				
customer_id	character varying(20)				
vehicle_type	character varying(50)				
cancelled_by_customer	integer				
customer_cancel_reason	text				
cancelled_by_driver	integer				
driver_cancel_reason	text				
incomplete_rides	integer				
incomplete_reason	text				
booking_value	numeric(10,2)				
ride_distance	numeric(10,2)				
driver_rating	numeric(2,1)				
customer_rating	numeric(2,1)				
payment_method	character varying(50)				

شکل ۱۰ ساختار جدول Bronze

در ادامه اسکیمای Silver و جدول silver_cleaned_dataset را می‌بینیم:

The screenshot shows a pgAdmin4 interface with the following details:

- Database:** silver_cleaned_dataset/uber_db/postgres@PostgreSQL 18
- Table:** silver_cleaned_dataset
- Rows:** Showing rows: 1 to 1000
- Page No:** 1 of 149
- Columns:** trip_timestamp, pickup_hour, day_of_week, day_name, month, year, is_weekend, unified_cancellation_reason, driver_rating, customer_rating, driver_rating_imputed, customer_rating_imputed, is_cancelled.
- Data:** The table contains 148767 rows of data, with the first few rows shown below.

	trip_timestamp	pickup_hour	day_of_week	day_name	month	year	is_weekend	unified_cancellation_reason	driver_rating	customer_rating	driver_rating_imputed	customer_rating_imputed	is_cancelled
1	2024-02-16 20:11:31	20	5	Friday	2	2024	false	Reason Not Specified	4.70	4.30	false	false	false
2	2024-11-21 10:30:43	10	4	Thursday	11	2024	false	Reason Not Specified	4.90	3.20	false	false	false
3	2024-03-28 11:57:01	11	4	Thursday	3	2024	false	Reason Not Specified	4.60	4.50	false	false	false
4	2024-04-12 19:48:39	19	5	Friday	4	2024	false	Reason Not Specified	[null]	[null]	false	false	true
5	2024-04-16 08:03:17	8	2	Tuesday	4	2024	false	Customer related issue	[null]	[null]	false	false	true
6	2024-03-16 11:12:04	11	6	Saturday	3	2024	true	Reason Not Specified	3.90	4.40	false	false	false
7	2024-08-16 17:35:55	17	5	Friday	8	2024	false	Reason Not Specified	4.60	4.70	false	false	false
8	2024-03-21 20:32:27	20	4	Thursday	3	2024	false	Reason Not Specified	3.10	4.60	false	false	false
9	2024-09-03 21:12:31	21	2	Tuesday	9	2024	false	Reason Not Specified	4.50	4.60	false	false	false
10	2024-11-29 18:21:41	18	5	Friday	11	2024	false	Reason Not Specified	3.80	4.20	false	false	false
11	2024-10-22 21:07:55	21	2	Tuesday	10	2024	false	Reason Not Specified	4.30	3.30	false	false	false
12	2024-02-16 10:01:46	10	5	Friday	2	2024	false	Change of plans	[null]	[null]	false	false	true
13	2024-07-02 20:30:03	20	2	Tuesday	7	2024	false	Personal & Car related issues	[null]	[null]	false	false	true
14	2024-07-06 09:09:09	9	6	Saturday	7	2024	true	Reason Not Specified	4.20	4.80	false	false	false
15	2024-03-19 09:46:34	9	2	Tuesday	3	2024	false	Reason Not Specified	4.40	4.10	false	false	false
16	2024-01-25 11:29:01	11	4	Thursday	1	2024	false	AC is not working	[null]	[null]	false	false	true
17	2024-09-03 07:49:31	7	2	Tuesday	9	2024	false	Personal & Car related issues	[null]	[null]	false	false	true
18	2024-12-11 12:14:05	12	3	Wednesday	12	2024	false	Reason Not Specified	4.50	4.50	false	false	false
19	2024-02-10 07:18:19	7	6	Saturday	2	2024	true	Reason Not Specified	4.10	4.90	false	false	false
20	2024-04-13 13:22:06	13	6	Saturday	4	2024	true	Reason Not Specified	4.20	4.30	false	false	false
21	2024-12-16 12:52:50	12	1	Monday	12	2024	false	Reason Not Specified	[null]	[null]	false	false	true
22	2024-03-01 11:17:51	11	5	Friday	3	2024	false	Reason Not Specified	4.10	4.90	false	false	false
23	2024-07-31 12:11:09	12	3	Wednesday	7	2024	false	Reason Not Specified	4.40	3.30	false	false	false
24	2024-06-20 08:09:13	8	4	Thursday	6	2024	false	Reason Not Specified	4.70	3.90	false	false	false
25	2024-10-04 20:37:55	20	5	Friday	10	2024	false	Customer Demand	[null]	[null]	false	false	true
26	2024-09-26 18:45:21	18	4	Thursday	9	2024	false	Reason Not Specified	4.00	4.20	false	false	false
27	2024-07-07 13:09:33	13	0	Sunday	7	2024	true	Reason Not Specified	[null]	[null]	false	false	true
28	2024-04-26 19:19:39	19	5	Friday	4	2024	false	Reason Not Specified	3.20	4.30	false	false	false
29	2024-09-09 13:57:31	13	1	Monday	9	2024	false	Reason Not Specified	4.70	4.50	false	false	false

شکل ۱۳ داده‌ها در لایه Silver بعد از تغییرات در pgAdmin4

خیلی واضح است که مقادیر Null بخوبی مدیریت شده و به شدت کاهش یافته‌اند. برای نشان دادن بهتر این موضوع ما یک کوئری پیچیده نوشته‌یم که مقادیر Null در دو لایه Bronze و Silver را به همراه درصد بھبود نشان می‌دهد که نتیجه آن را اینجا می‌اوریم:

```
uber_db=#      s.total_null_values AS "Null Count",
uber_db=#      (b.total_null_values - s.total_null_values) AS "Nulls Removed",
uber_db=#      ROUND(((b.total_null_values - s.total_null_values)::NUMERIC / NULLIF(b.total_null_values, 0)) * 100, 2)
ovement"
uber_db# FROM bronze_stats b, silver_stats s;
   Layer | Null Count |          Layer          | Null Count | Nulls Removed | Data Quality Improvement
-----+-----+-----+-----+-----+-----+
Bronze (Raw Data) | 1065000 | Silver (Cleaned Data) | 313500 |    751500 | 70.56%
(1 row)
```

شکل ۱۴ مقایسه null value ها در دو لایه نقره و برنز و درصد بھبود

با توجه به حجم دیتا (۱۵۰,۰۰۰ ردیف) و تعداد ستون‌ها (حدود ۱۷ ستون در برنز)، کاهش به ۳۱۳ هزار یعنی میانگین NULL به ازای هر ردیف از حدود ۷ به ۲ رسیده. این نشون میده پالایش داده‌ها خیلی مؤثر بوده است. و نسبت بھبود ۷۰٪ گویای این موضوع است.

booking_id	trip_timestamp	pickup_hour	day_name	is_weekend	unified_cancellation_reason	driver_rating	driver_rating_imputed	customer_rating	customer_rating_imputed	s_cancelled
"CNR5884380"	2024-03-23 12:29:38	12	Saturday	t	Reason Not Specified	f	f	f	f	
"CNR1368890"	2024-03-29 18:01:15	18	Friday	f	Vehicle Breakdown	f	f	f	f	
"CNR5138897"	2024-09-09 17:09:29	15	Wednesday	t	Reason Not Specified	f	f	f	f	
"CNR721897"	2024-12-16 19:06:08	19	Monday	f	Other Issues	f	f	f	f	
"CNR8551927"	2024-09-18 08:09:38	8	Wednesday	f	Reason Not Specified	f	f	f	f	
"CNR4386945"	2024-06-25 22:44:15	22	Tuesday	f	Personal & Car related issues	f	f	f	f	
"CNR6739317"	2024-12-15 15:08:25	15	Sunday	t	Customer related issue	f	f	f	f	
"CNR6126048"	2024-11-24 09:07:10	9	Sunday	t	Driver is not moving towards pickup location	f	f	f	f	
"CNR9465848"	2024-01-20 19:53:57	19	Friday	f	Customer related issue	f	f	f	f	
"CNR3510850"	2024-01-15 10:49:09	9	Friday	f	Personal & Car related issues	f	f	f	f	
"CNR4213833"	2024-04-12 07:42:35	19	Friday	f	Reason Not Specified	f	f	f	f	
"CNR9834281"	2024-04-05 18:57:12	18	Friday	f	Vehicle Breakdown	f	f	f	f	
"CNR2178654"	2024-05-18 17:37:52	17	Saturday	t	More than permitted people in there	f	f	f	f	
"CNR8807055"	2024-07-13 19:53:15	19	Saturday	t	Customer related issue	f	f	f	f	
"CNR4213847"	2024-09-18 13:02:42	13	Tuesday	f	Driver is not moving towards pickup location	f	f	f	f	
(15 rows)										

شکل ۱۵ جدول Silver و مدیریت مقادیر Null

بررسی عدم وجود Null در ستون جدید unified_cancellation_reason با یک کوئری ساده:

```
SELECT
    COUNT(*) AS total_rows,
    COUNT(*) FILTER (WHERE unified_cancellation_reason IS NULL) AS null_count
FROM silver.cleaned_dataset;
```

total_rows	null_count
148767	0

شکل ۱۶ عدم وجود Null در ستون جدید unified_cancellation_reason

در ادامه نیز دلایل لغو رو گزارش می‌کنیم:

```
SELECT
    unified_cancellation_reason,
    COUNT(*) AS frequency
FROM silver.cleaned_dataset
GROUP BY unified_cancellation_reason
ORDER BY frequency DESC;
```

unified_cancellation_reason	frequency
Reason Not Specified	102649
Customer related issue	6777
The customer was coughing/sick	6693
Personal & Car related issues	6675
More than permitted people in there	6644
Customer Demand	3015
Vehicle Breakdown	2993
Other Issue	2919
Wrong Address	2348
Change of plans	2326
Driver is not moving towards pickup location	2315
Driver asked to cancel	2274
AC is not working	1139
(13 rows)	

شکل ۱۷ دلایل لغو سفر و تعداد هر کدام به صورت یکپارچه

اما چون براساس نوع خودرو میانگین متفاوتی رو جایگزین می‌کردیم یک کوئری به تفکیک نوع خودرو می‌نویسیم:

```
SELECT
    vehicle_type,
    ROUND(AVG(driver_rating)::NUMERIC, 2) AS avg_driver,
    ROUND(AVG(customer_rating)::NUMERIC, 2) AS avg_customer,
    COUNT(*) AS trip_count
FROM silver.cleaned_dataset
WHERE booking_status = 'Completed'
GROUP BY vehicle_type
ORDER BY avg_driver DESC;
```

vehicle_type	avg_driver	avg_customer	trip_count
Uber XL	4.24	4.40	2765
Bike	4.23	4.40	13921
eBike	4.23	4.40	6480
Auto	4.23	4.40	22970
Go Sedan	4.23	4.41	16550
Premier Sedan	4.23	4.40	11158
Go Mini	4.23	4.40	18404
(7 rows)			

شکل ۱۸ میانگین امتیازها با توجه به نوع خودرو

۲-۶- بررسی تعداد رکوردها و مدیریت داده‌های تکراری

پس از اجرای عملیات ETL و انتقال داده‌ها از لایه برنز به لایه نقره‌ای، متوجه شدم که تعداد رکوردهای جدول نقره‌ای (silver.cleaned_dataset) با تعداد رکوردهای جدول برنز (bronze.raw_dataset) برابر نیست. جدول برنز ۱۵۰,۰۰۰ رکورد داشت در حالی که جدول نقره‌ای ۱۴۸,۷۶۷ رکورد، یعنی ۱,۲۳۳ رکورد کمتر.

برای پیدا کردن علت، کوئری زیر را اجرا کردم تا ببینم آیا booking_id تکراری وجود دارد یا خیر؟

```
uber_db=# SELECT booking_id, COUNT(*)
uber_db=# FROM bronze.raw_dataset
uber_db=# GROUP BY booking_id
uber_db=# HAVING COUNT(*) > 1;
   booking_id | count
-----+-----
  "CNR9555508" | 2
  "CNR9027093" | 2
  "CNR5598384" | 2
  "CNR4042457" | 2
  "CNR2078514" | 2
  "CNR5627941" | 2
  "CNR7096387" | 2
  "CNR5341647" | 2
  "CNR1056023" | 2
  "CNR5358950" | 2
```

شکل ۱۹ بررسی وجود booking_id تکراری

دلیل این اتفاق در انبار داده (Data Warehouse) این است که، ستون booking_id باید به عنوان کلید اصلی (Primary Key) عمل کند و مقدار یکتا داشته باشد. وجود رکوردهای تکراری می‌تواند دلایلی مانند: خطا در فرایند ثبت داده یا ممکن است داده‌ها دو بار از منبع اصلی دریافت شده باشند یا مشکل از منبع داده باشد. رویکرد من در مدیریت این موضوع این است که:

با توجه به اینکه در لایه نقره‌ای می‌خواهیم داده‌هایی تمیز، یکتا و آماده تحلیل داشته باشیم، تصمیم گرفتم از ON CONFLICT (booking_id) DO NOTHING استفاده کنم. این دستور تضمین می‌کند که: فقط اولین نسخه از هر booking_id وارد جدول نقره شود و از ایجاد رکورد تکراری در جدول نقره جلوگیری شود و مهم‌تر از همه کلید اصلی یکتا بماند.

```
uber_db=# SELECT COUNT(*) FROM bronze.raw_dataset;
   count
-----
 150000
(1 row)

uber_db=# SELECT COUNT(*) FROM silver.cleaned_dataset;
   count
-----
 148767
(1 row)
```

شکل ۲۰ مقایسه تعداد رکوردهای جدول Bronze و Silver

گزارش فاز چهارم: Gold Medalion Architecture لایه

اکنون که با موفقیت لایه نقره‌ای را تکمیل کردیم می‌رویم به سراغ لایه طلای (Gold). توی این لایه قراره داده‌ها رو برای مصرف نهایی آماده کنیم؛ تمیزترین و غنی‌ترین نسخه، به همراه ستون‌های محاسباتی، محدودیت‌های یکپارچگی و کلید اصلی. این لایه مستقیماً در داشبوردها، API و دستیارهای هوشمند Text-To-SQL استفاده خواهد شد. طبق صورت پروژه، سه اقدام اصلی داریم:

- **طراحی Schema نهایی** با نام gold.dataset و انتخاب نوع‌های داده بهینه. از NUMERIC برای دقت مالی و TEXT برای رشته‌ها که انعطاف‌پذیرتر هست استفاده کردم.
- **محاسبه و غنی‌سازی**: اضافه کردن ستون محاسباتی مثل درآمد به ازای هر کیلومتر (revenue_per_km) و ...
- **یکپارچگی داده**: تعیین Primary Key و اضافه کردن محدودیت CHECK برای اطمینان از بازه‌ی معتبر امتیازها (مثلاً بین ۰ تا ۵).

علاوه بر این، باید مطمئن بشیم که تمیزترین نسخه داده‌ها رو توی این جدول داریم. از اونجایی که لایه نقره‌ای قبل‌اً تمیز و غنی شده، کار زیادی برای پاکسازی نمی‌مونه، اما می‌توانیم مطمئن بشیم که امتیازها در بازه درست هستن و محاسبات به درستی انجام می‌شوند.

مثل قبل اسکریپت رو اینطوری ساختم که همه چیز در یک تراکنش باشه تا اگر مشکلی پیش اوهد، هیچ تغییری ذخیره نشه و دیتابیس مشکلی پیدا نکنه.

Insert اصلی رو نوشتم که داده‌ها رو از silver می‌کشه و revenue_per_km رو محاسبه می‌کنه گذاشتم تا بگویم اگر مسافت بیشتر از صفر بود درآمد به ازای کیلومتر حساب بشه، و گرنه null بمونه. (مثل سفرهای لغو شده)، چراکه تقسیم بر صفر خطأ می‌ده و منطقی نیست درآمد برای سفر بدون مسافت حساب کنیم:

```
CASE
    WHEN ride_distance > 0 THEN booking_value / ride_distance
    ELSE NULL
END AS revenue_per_km,
```

بقیه ستون‌ها رو مستقیم کپی کردم چون در Silver تمیز شدن، flags و imputed هارو و is_cancelled ز رو هم نگه داشتم تا در داشبورد بتونیم فیلتر کنیم که کدوم‌ها لغو شدن.

```
ggr@DESKTOP-M04J70A MINGW64 /d/Daneshgoh/Master/Term 1/Advance Data Base/Tamrin/Project/uber_data_platform/backend/src
$ node gold_etl.js
[dotenv@17.2.4] injecting env (5) from ..\env -- tip: ♦ encrypt with Dotenvx: https://dotenvx.com
[dotenv@17.2.4] injecting env (0) from ..\..\env -- tip: ♦ prevent building .env in docker: https://dotenvx.com/prebuild
♦ Connected to PostgreSQL database
♦ Starting Gold Layer ETL ...
♦ Gold table schema created/verified
✓ Data transferred to Gold with revenue_per_km enriched
♦ Gold Layer ETL completed successfully!
```

شکل ۲۱ خروجی موفق عملیات Gold

Column	Type	Collation	Nullable	Default
booking_id	text		not null	
trip_timestamp	timestamp without time zone			
pickup_hour	integer			
day_name	text			
is_weekend	boolean			
booking_status	text			
customer_id	text			
vehicle_type	text			
unified_cancellation_reason	text			
booking_value	numeric(10,2)			
ride_distance	numeric(10,2)			
revenue_per_km	numeric(10,2)			
driver_rating	numeric(3,2)			
customer_rating	numeric(3,2)			
payment_method	text			
driver_rating_imputed	boolean			
customer_rating_imputed	boolean			
is_cancelled	boolean			

Indexes:

- "gold_dataset_pkey" PRIMARY KEY, btree (booking_id)

Check constraints:

- "gold_dataset_customer_rating_check" CHECK (customer_rating >= 0::numeric AND customer_rating <= 5::numeric)
- "gold_dataset_driver_rating_check" CHECK (driver_rating >= 0::numeric AND driver_rating <= 5::numeric)

شکل ۲۲ ساختار و اسکیمای جدول Gold

در پایان ما به اطلاعات مفیدی میانگین درآمد هر سفر به ازای هر کیلومتر:

```
SELECT
    COUNT(*) AS total_rows,
    COUNT(CASE WHEN revenue_per_km IS NULL THEN 1 END) AS null_revenue_per_km,
    ROUND(AVG(revenue_per_km), 2) AS avg_revenue_per_km
FROM gold.gold_dataset;
```

```
uber_db=# SELECT
uber_db#   COUNT(*) AS total_rows,
uber_db#   COUNT(CASE WHEN revenue_per_km IS NULL THEN 1 END) AS null_revenue_per_km,
uber_db#   ROUND(AVG(revenue_per_km), 2) AS avg_revenue_per_km
uber_db# FROM gold.gold_dataset;
total_rows | null_revenue_per_km | avg_revenue_per_km
-----+-----+-----
 148767 |          47592 |        37.83
(1 row)
```

میانگین درآمد به تفکیک وسیله نقلیه:

```
SELECT
    vehicle_type,
    COUNT(*) AS ride_count,
    ROUND(AVG(revenue_per_km), 2) AS avg_revenue_per_km
FROM gold.gold_dataset
WHERE revenue_per_km IS NOT NULL
GROUP BY vehicle_type
ORDER BY avg_revenue_per_km DESC;
```

vehicle_type	ride_count	avg_revenue_per_km
Go Sedan	18181	38.23
Bike	15239	38.14
Auto	25218	38.08
Uber XL	3023	37.69
Premier Sedan	12208	37.62
Go Mini	20198	37.61
eBike	7108	36.33
(7 rows)		

شکل ۲۳ میانگین درآمد در هر کیلومتر به تفکیک نوع وسیله نقلیه

The screenshot shows a PostgreSQL database connection named 'gold.gold_dataset/uber_db/postgres@PostgreSQL 18'. The table 'uber' is displayed with 149 rows. The columns are: id, day_name, is_weekend, booking_status, customer_id, vehicle_type, unified_cancellation_reason, booking_value, ride_distance, revenue_per_km, driver_rating, customer_rating, and payment_method. The data includes various Uber trips with different vehicle types, cancellation reasons, and payment methods like UPI and Uber Wallet.

شکل ۲۴ جدول Gold

در نهایت یک مقایسه کوتاهی از سه لایه داشته باشیم:

جدول ۲ مقایسه سه لایه معماری مدل‌الیون

ویژگی	Bronze (Raw)	Silver (Cleaned)	Gold (Business Ready)
تعداد ستون	۱۷	۲۱ (با ویژگی‌های جدید)	۱۸ (بهینه‌شده)
revenue_per_km	ندارد	ندارد	دارد (با مدیریت صفر)
unified_cancellation_reason	ندارد	دارد	دارد (بهینه)
Primary Key	ندارد	دارد	دارد
revenue_per_kmNotNull	—	—	فقط در سفرهای لغو یا ناقص
آماده برای داشبورد	خیر	نیمه آماده	کاملاً آماده

گزارش فاز پنجم: پیاده‌سازی CRUD API

در این بخش ما به پیاده‌سازی بکاند (backend) (express.js) به کمک express.js و محیط توسعه Node.js پرداختیم و سپس با استفاده از REST-full API's و ساخت endpoint های مختلف برای فرانت‌اند توانستیم اطلاعات ذخیره شده داخل دیتابیس را در یک داشبورد عملیاتی به صورت ویژوال و با UI/UX مناسب نمایش بدهیم. (این در بخش بعد به طول کامل بررسی می‌شود).

۱-۲- ساختار تعریف شده :Backend

تویی یه پروژه بکاند با Express ، ما کد رو به بخش‌های مختلف تقسیم می‌کنیم تا تمیز و قابل فهم باشه:

- (مسیرها): اینجا مشخص می‌کنیم که برای هر آدرس (URL) و متد (GET, POST, PUT, DELETE...) کد کجا باید اجرا بشه. مثلًاً اگه کاربر به GET درخواست /api/trips را بده، باید بره تابع getTrips رو اجرا کنه
- (کنترلرها): اینجا توابع اصلی رو می‌نویسیم که کار واقعی رو انجام می‌دن (مثل کوئری زدن به دیتابیس، پردازش داده، برگرداندن نتیجه و ...) - tripController در برنامه این کار رو انجام می‌دهد.
- (مدل‌ها): اینجا معمولاً کدهایی برای ارتباط با دیتابیس می‌ذاریم (مثل اتصال pool - db.js) در برنامه این کار رو انجام می‌دهد.
- این جداسازی باعث می‌شه اگه بعداً خواستیم یه چیزی رو عوض کنیم، راحتتر باشه. مثلًاً اگه مسیر یه API عوض بشه، فقط فایل routes رو تغییر می‌دیم و کاری به کنترلر نداریم

برای پیاده‌سازی عملی و اصولی ما ۳ مرحله داریم:

۱. نصب های لازم: که بعدا در فایل packages.json می‌بینیم.

۲. ایجاد اتصال بین دیتابیس و برنامه:

```
const { Pool } = require("pg");
require("dotenv").config({ path: "../../.env" });
const pool = new Pool({
  user: process.env.DB_USER,
  host: process.env.DB_HOST,
  database: process.env.DB_NAME,
  password: process.env.DB_PASSWORD,
  port: process.env.DB_PORT,
});
pool.on("connect", () => {
  console.log(`Connected to PostgreSQL database`);
});
pool.on("error", (err) => {
  console.error(`X Unexpected error on idle client`, err);
  process.exit(-1);
});
module.exports = pool
```

۳. ساخت سرور بک اند (run server) و مدیریت route‌ها:

```
require('dotenv').config({ path: './.env' });
const express = require('express');
const cors = require('cors');
const app = express();
// Middleware for read request
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(cors()); // because we have separated frontend
const tripRoutes = require('./routes/trips');
app.use('/api/trips', tripRoutes);
const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

۴. پیاده‌سازی عملیات CRUD در کنترلرها و تست از طریق ابزار (در اینجا از postman استفاده شده)

۲-۲ - عملیات CRUD

پس از پیاده‌سازی فانکشن‌ها و عملیات‌ها ما توسط ۴ اندپوینت، عملکردهای خواسته شده در صورت پروژه رو برآورده می‌کنیم:

```
const tripController = require("../controllers/tripController");

router.post("/", tripController.createTrip);
router.get("/", tripController.getTrips);
router.patch("/:id", tripController.updateTripStatus);
router.delete("/:id", tripController.deleteTrip);
```

CREATE - Post API - ۲-۳-۱

The screenshot shows a Postman interface with the following details:

- Method:** POST
- URL:** http://localhost:5000/api/trips
- Body:** JSON (selected)
- Request Body (JSON):**

```

1 {
2   "date": "2030-02-28",
3   "time": "14:45:00",
4   "customer_id": "CIDTEST999",
5   "vehicle_type": "Go Sedan",
6   "booking_value": 650,
7   "ride_distance": 18.7,
8   "payment_method": "UPI",
9   "driver_rating": 4.7,
10  "customer_rating": 4.9
11 }

```

- Response Headers:** Status: 201 Created, Time: 53 ms, Size: 808 B
- Response Body (Pretty JSON):**

```

1 {
2   "message": "Trip created successfully",
3   "trip": {
4     "booking_id": "CNR1831406",
5     "trip_timestamp": "2030-02-28T11:15:00.000Z",
6     "pickup_hour": null,
7     "day_name": null,
8     "is_weekend": null,
9     "booking_status": "Completed",
10    "customer_id": "CIDTEST999",
11    "vehicle_type": "Go Sedan",
12    "unified_cancellation_reason": "Reason Not Specified",
13    "booking_value": "650.00",
14    "ride_distance": "18.70",
15    "revenue_per_km": "34.76",
16    "driver_rating": "4.70"
}

```

شکل ۲۵ - ساخت یک سفر جدید

در اینجا همانطور که مشاهده می‌کنیم از طریق روت POST /api/trips، ما اطلاعات ضروری تعریف شده را از طریق body و به صورت JSON به این روت می‌فرستیم و این اطلاعات تحت عنوان یک سفر جدید داخل پایگاه داده با موفقیت ذخیره می‌شوند.

فرض بر این بوده که وضعیت سفر booking_id به صورت خودکار تولید شود. بقیه اطلاعات مانند اطلاعات تاریخ، زمان، نوع خودرو و روش پرداخت و امتیازها از کاربر به صورت Optional دریافت شود.

READ - GET API - ۲-۳-۲

در اینجا از طریق متد GET و از طریق روت /api/trips می‌توانیم به دو صورت عمل کنیم:

۱. همه رکوردها رو برگردانیم.

۲. یک رکورد خاص، رو بتوانیم از طریق customer_id یعنی شناسه مشتری برگردانیم.

```

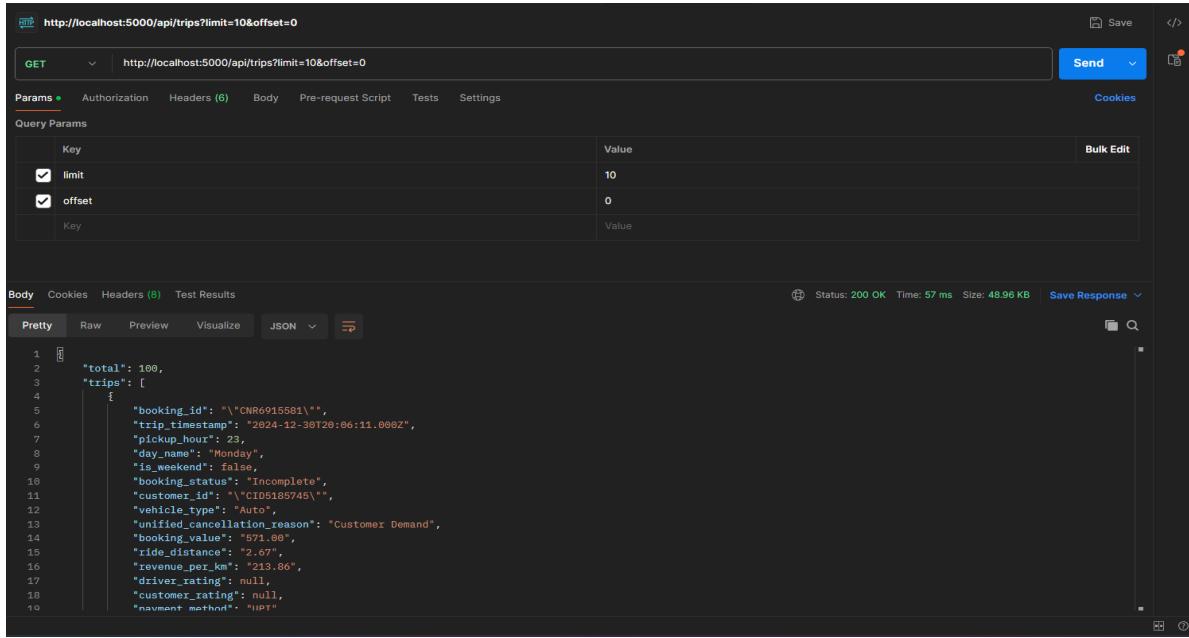
1
2   "total": 1,
3   "trips": [
4     {
5       "booking_id": "CNR1831486",
6       "trip_timestamp": "2030-02-20T11:15:00.000Z",
7       "pickup_hour": null,
8       "day_name": null,
9       "is_weekend": null,
10      "booking_status": "Completed",
11      "customer_id": "CIDTEST999",
12      "vehicle_type": "Go Sedan",
13      "unified_cancellation_reason": "Reason Not Specified",
14      "booking_value": "650.00",
15      "ride_distance": "18.76",
16      "revenue_per_km": "34.76",
17      "driver_rating": "4.70",
18      "customer_rating": "4.90",
19      "payment_method": "UPI",
20      "driver_rating_imputed": false,
21      "customer_rating_imputed": false,
22      "is_cancelled": false
    }
  ]
}

```

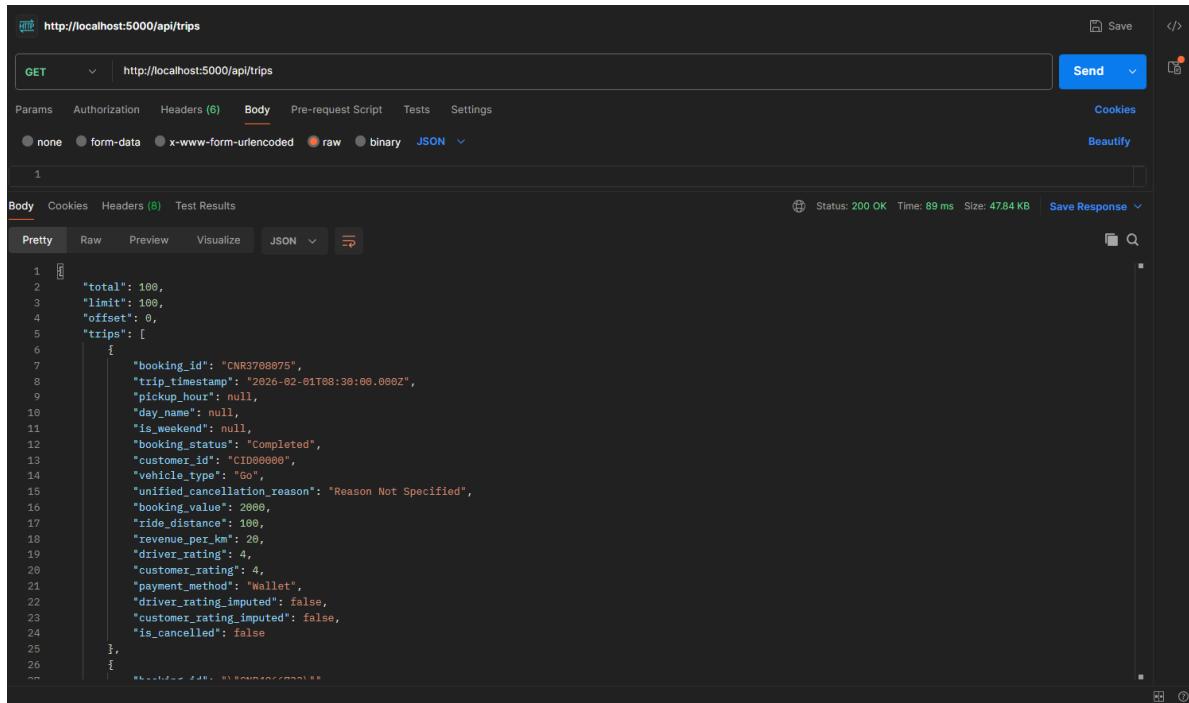
شکل ۲۶ - خواندن یک رکورد با customer_id مشخص

کافی است customer_id را از طریق Query Params دریافت کنیم و داخل پایگاهداده سرچ بزنیم.
همانطور که مشاهده می‌شود رکورد مورد نظر پیدا شده است.

حالی که بتوانیم تمامی رکوردها رو بدون هیچ شرطی بخوانیم هم محیا هست اما برای کنترل بیشتر می‌توانیم داخل Query Params لیمیت و آفست تنظیم کنیم:



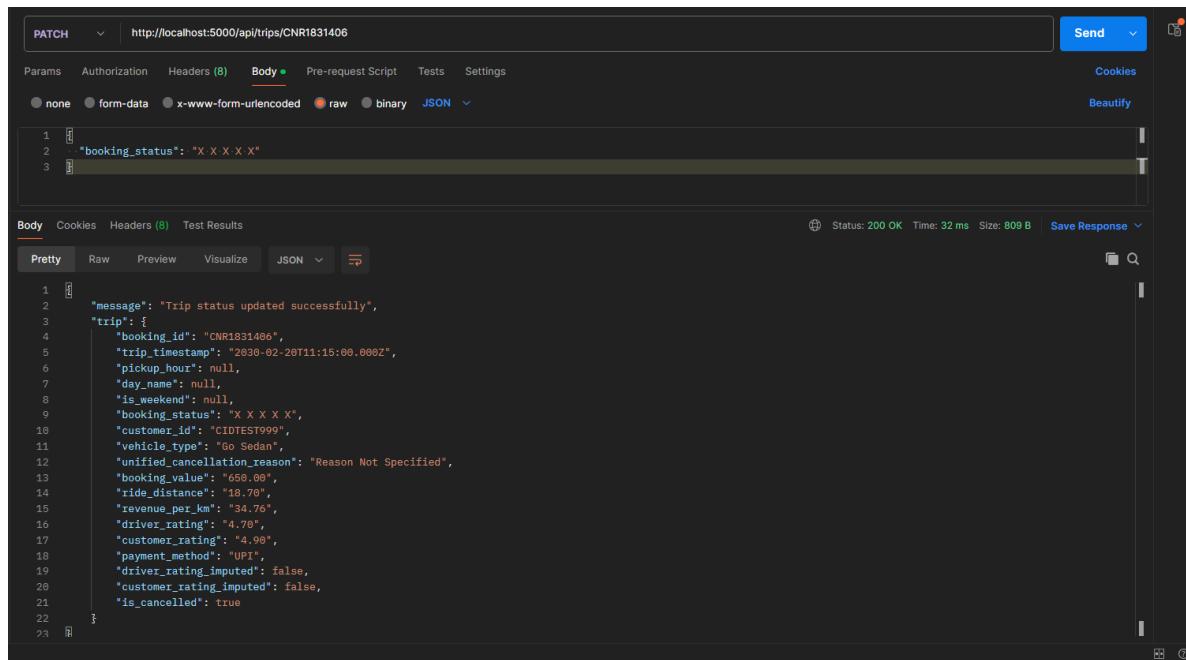
شکل ۲۷ - خواندن همه رکوردها یا اعمال فیلتر با کمک `limit` و `offset`



شکل ۲۸ - خواندن همه رکوردها بدون هیچ فیلتری

UPDATE - PATCH API - ۲-۳-۳

برای بروزرسانی یک سفر هم کافی است آیدی آن رزرو رو در URL بفرستیم و همچنین وضعیت مورد نظرمان در مورد آن سفر رو هم در Body ارسال کنیم تا بروزرسانی ما انجام شود:



The screenshot shows a POSTMAN interface with the following details:

- Method:** PATCH
- URL:** http://localhost:5000/api/trips/CNR1831406
- Body (JSON):**

```

1
2   "booking_status": "X X X X X"
3
        
```
- Response:**

```

1
2   "message": "Trip status updated successfully",
3   "trip": [
4     {
5       "booking_id": "CNR1831406",
6       "trip_timestamp": "2020-02-28T11:15:00.000Z",
7       "pickup_hour": null,
8       "day_name": null,
9       "is_weekend": null,
10      "booking_status": "X X X X X",
11      "customer_id": "CIDTEST0999",
12      "vehicle_type": "Go Sedan",
13      "unified_cancellation_reason": "Reason Not Specified",
14      "booking_value": "650.00",
15      "ride_distance": "18.79",
16      "revenue_per_km": "34.76",
17      "driver_rating": "4.78",
18      "customer_rating": "4.00",
19      "payment_method": "UPI",
20      "driver_rating_imputed": false,
21      "customer_rating_imputed": false,
22      "is_cancelled": true
23    }
  
```

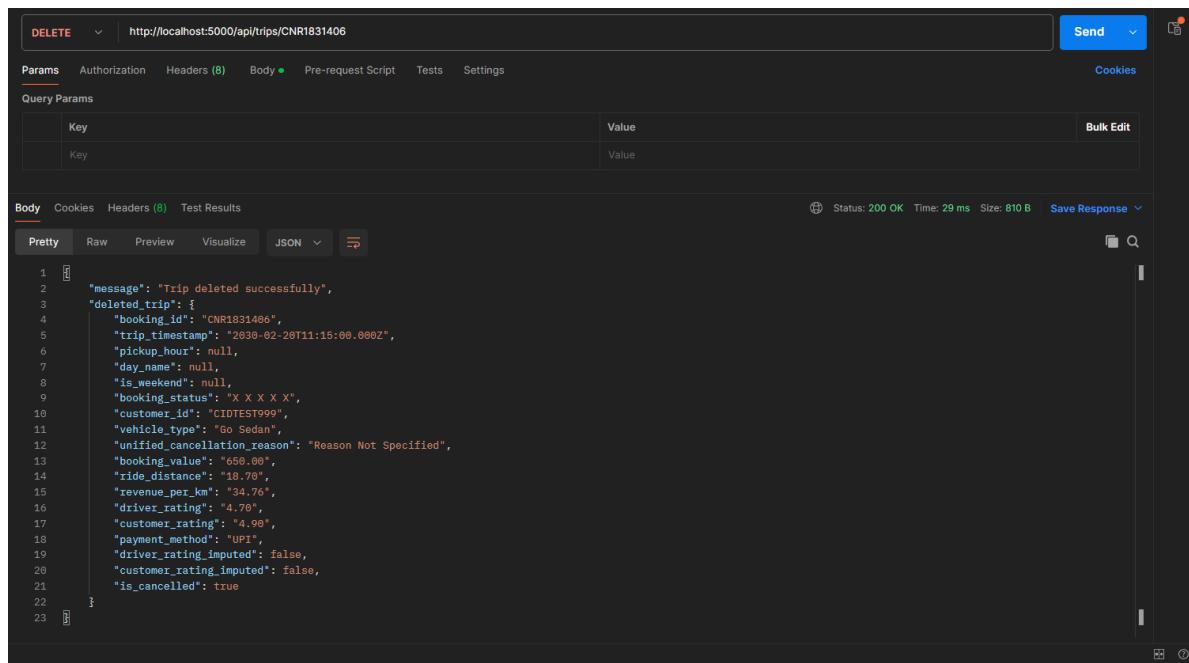
شکل ۲۹ - بروزرسانی وضعیت یک سفر از طریق **booking_id**

همانطور که مشاهده می‌شود: ما وضعیت سفر رو با کمک آیدی: CNR1831406 به مقدار دلخواه:

تغییر داده‌ایم. Booking_status : “X X X X X”

DELETE API - ۲-۳-۴

ما از همین آیدی CNR1831406 استفاده می‌کنیم و این سفر را حذف کنیم:



The screenshot shows a Postman interface with the following details:

- Method:** DELETE
- URL:** <http://localhost:5000/api/trips/CNR1831406>
- Headers:** Authorization, Headers (8), Body, Pre-request Script, Tests, Settings, Cookies
- Query Params:** Key, Value
- Body:** Pretty, Raw, Preview, Visualize, JSON
- Response Status:** 200 OK, Time: 29 ms, Size: 810 B, Save Response
- Response Content (Pretty Print):**

```

1  "message": "Trip deleted successfully",
2  "deleted_trip": {
3    "booking_id": "CNR1831406",
4    "trip_timestamp": "2030-02-20T11:15:00.000Z",
5    "pickup_hour": null,
6    "day_name": null,
7    "is_weekend": null,
8    "booking_status": "X X X X",
9    "customer_id": "CIDTEST999",
10   "vehicle_type": "Go Sedan",
11   "unified_cancellation_reason": "Reason Not Specified",
12   "booking_value": "650.00",
13   "ride_distance": "18.70",
14   "revenue_per_km": "34.76",
15   "driver_rating": "4.70",
16   "customer_rating": "4.90",
17   "payment_method": "UPI",
18   "driver_rating_imputed": false,
19   "customer_rating_imputed": false,
20   "is_cancelled": true
21 }
22
23

```

شکل ۳۰ - حذف یک سفر

گزارش فاز ششم : داشبورد تحلیلی و تعاملی

در این بخش هدف من این بود که داده‌های تمیز و مدل‌سازی‌شده‌ی لایه Gold را به یک داشبورد تحلیلی تبدیل کنم که هم از نظر بصری حرفه‌ای باشد و هم از نظر منطقی، پاسخ‌گوی نیازهای تحلیلی پروژه باشد. تمرکز اصلی من این بود که تمام شاخص‌ها و نمودارها به صورت داینامیک و وابسته به فیلترها کار کنند و صرفاً یک صفحه‌ی نمایشی ساده نباشد.

۲-۱- طراحی معماری سمت بک‌اند برای داشبورد

برای اینکه داشبورد واقعاً تحلیلی باشد، تصمیم گرفتم منطق تجمعی (Aggregation) را در سمت دیتابیس انجام دهم، نه در فرانت‌اند. دلیل این تصمیم این بود که دیتابیس برای عملیات GROUP BY و SUM و COUNT بهینه‌تر است همچنین از انتقال حجم زیاد داده‌ی خام به فرانت جلوگیری می‌شود و ساختار پروژه حرفه‌ای‌تر و مقیاس‌پذیرتر می‌شود.

برای این منظور، چند endpoint جداگانه برای داشبورد طراحی کردم:

- دریافت شاخص‌های کلیدی عملکرد(KPI¹s)
- دریافت داده‌های نمودارها
- دریافت لیست سفرها با pagination و فیلتر
- پشتیبانی کامل از فیلتر بازه زمانی و نوع خودرو

تمام این API‌ها از پارامترهای زیر پشتیبانی می‌کنند:

- startDate
- endDate
- vehicleType

¹: شاخص کلیدی عملکرد Key Performance Indicator

به این صورت، هر بار که کاربر فیلتر را تغییر می‌دهد، داده‌های داشبورد به صورت کامل به روزرسانی می‌شوند. تقریباً در تمام توابع تحلیلی، یک الگوی مشترک رعایت شده است:

ابتدا فیلترها از req.query دریافت می‌شوند:

```
const { startDate, endDate, vehicleType } = req.query;
```

سپس سه متغیر برای ساخت شرط‌های پویا تعریف می‌شود:

```
let whereClauses = [];
let values = [];
let idx = 1;
```

▪ whereClauses: برای نگهداری شرط‌ها

▪ values: برای مقادیر پارامتری

▪ idx: برای شماره‌گذاری پارامترها (...).

به عنوان مثال اگر کاربر تاریخ شروع را ارسال کرده باشد، شرط زیر اضافه می‌شود:

```
trip_timestamp >= $1
```

و مقدار آن در آرایه values ذخیره می‌شود. این روش باعث می‌شود کوئری‌ها به صورت Parameterized SQL Injection جلوگیری شود.

در نهایت اگر شرطی وجود داشته باشد، بخش WHERE به صورت پویا ساخته می‌شود و به کوئری اصلی اضافه می‌گردد.

```
const whereSQL = whereClauses.length > 0 ? `WHERE ${whereClauses.join(' AND ')}` : '';
```

۲-۲- مدیریت ایمن مقادیر عددی

از آنجا که خروجی‌های PostgreSQL ممکن است به صورت رشته یا مقدار null برگردند، تابعی به نام safeNumber تعریف شد. این تابع ابتدا بررسی می‌کند که مقدار null نباشد، سپس آن را به عدد تبدیل می‌کند و در صورتی که تبدیل نامعتبر باشد (NaN شود)، مقدار پیش‌فرض برمی‌گرداند. به این ترتیب داده‌های ارسال شده به فرانت‌اند همیشه عددی و قابل استفاده در نمودارها هستند و از بروز خطا جلوگیری می‌شود.

```

3 // Helper function to safely parse numeric values
4 const safeNumber = (value, fallback = null) => {
5   if (value == null) return fallback;
6   const num = Number(value);
7   return Number.isNaN(num) ? fallback : num;
8 };

```

شکل ۳۱ کد تابع safeNumber برای مدیریت مقادیر عددی

۲-۳- شاخص‌های کلیدی عملکرد (KPI)

در این قسمت، چهار شاخص اصلی پیاده‌سازی شد:

- تعداد کل رزروها
- تعداد رزروهای موفق
- مجموع درآمد
- نرخ موفقیت سفر

منطق این بخش به صورت مستقیم داخل یک کوئری تجمعی در PostgreSQL نوشته شد. به طور خلاصه، از COUNT و SUM و FILTER استفاده کردم تا بتوانم همزمان چند شاخص را از یک اسکن دیتاباس استخراج کنم.

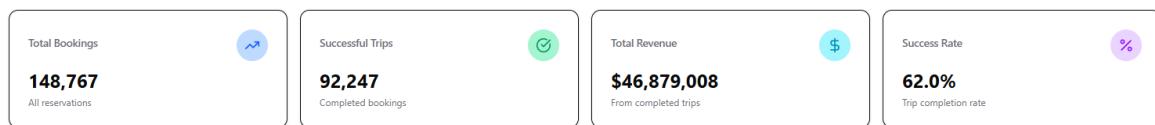
نمونه‌ای از منطق استفاده شده:

```

const query = `

SELECT
  COUNT(*) AS total_bookings,
  COUNT(*) FILTER (WHERE is_cancelled = FALSE) AS successful_bookings,
  SUM(booking_value) FILTER (WHERE is_cancelled = FALSE) AS total_revenue,
  ROUND(
    (COUNT(*) FILTER (WHERE is_cancelled = FALSE)::NUMERIC / NULLIF(COUNT(*), 0)) * 100, 2) AS
    success_rate
FROM gold.gold_dataset
${whereSQL}
`;

```

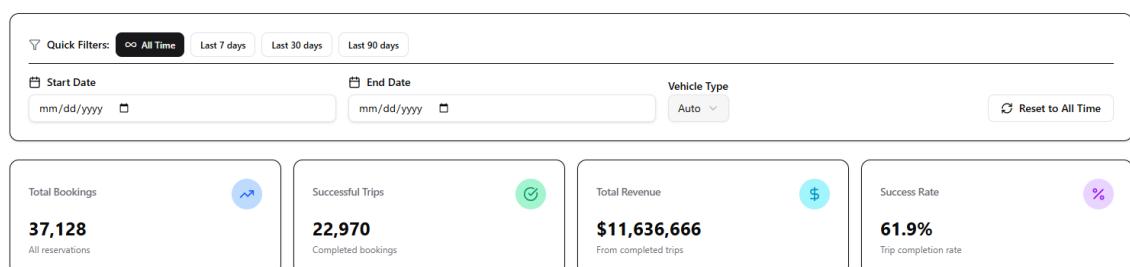


شکل ۳۲ نمایش KPI ها در داشبورد

همچنین ما می‌توانیم با تعیین پارامترها و فیلترهای اضافه مانند فیلترزمانی مثل: تاریخ شروع و تاریخ پایان یا فیلتر بر اساس نوع خودرو، KPI های مربوطه را خیلی سریع مشاهده کنیم:

Analytics Dashboard

Comprehensive trip performance and revenue analysis

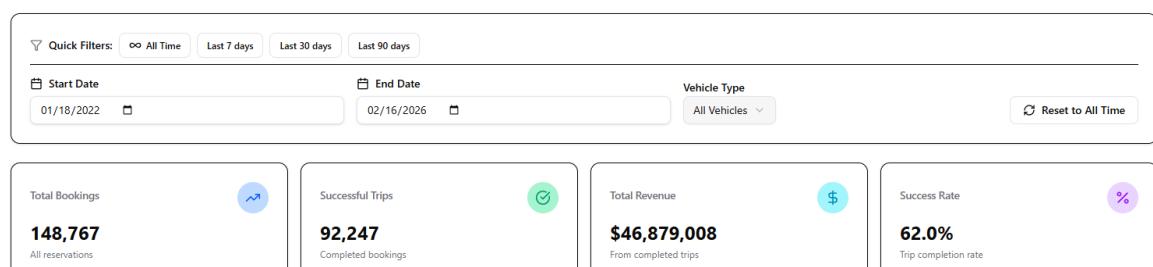


شکل ۳۳ نمایش KPI ها با فیلتر نوع خودرو

همچنین فیلترهای آمده‌ای مانند ۷ روز اخیر، یک ماه اخیر و ۹۰ روز اخیر به صورت خیلی سریع در دسترس قرار دارند.

Analytics Dashboard

Comprehensive trip performance and revenue analysis



شکل ۳۴ نمایش KPI ها با فیلتر زمانی

۲-۴ - نمودارها (Charts)

در این بخش، برای هر نمودار یک Endpoint مجزا در بکاند پیاده‌سازی شد که داده‌ها را از جدول استخراج کرده و پس از تجمعی، به فرانت‌اند ارسال می‌کند.

تمام نمودارها قابلیت اعمال فیلتر بازه زمانی (startDate, endDate) و نوع خودرو (vehicleType) را دارند که به صورت پویا در بخش WHERE کوئری اضافه می‌شوند.

۲-۳-۱ - نمودار دایره‌ای توزیع دلایل لغو

برای تحلیل علل شکست سفرها، از ستون unified_cancellation_reason استفاده کردم. در ابتدای تابع مربوطه (getCancellationReasons) شرط زیر به صورت پیش‌فرض در WHERE قرار داده شده است:

```
let whereClauses = ['is_cancelled = TRUE'];
```

یعنی فقط سفرهای لغو شده بررسی می‌شوند. سپس کوئری اصلی به شکل زیر نوشته شده است:

```
const query = `SELECT
    unified_cancellation_reason AS reason,
    COUNT(*) AS count,
    ROUND((COUNT(*)::NUMERIC / SUM(COUNT(*)) OVER ()) * 100, 2) AS percentage
FROM gold.gold_dataset
${whereSQL}
GROUP BY unified_cancellation_reason
ORDER BY count DESC
`;
```

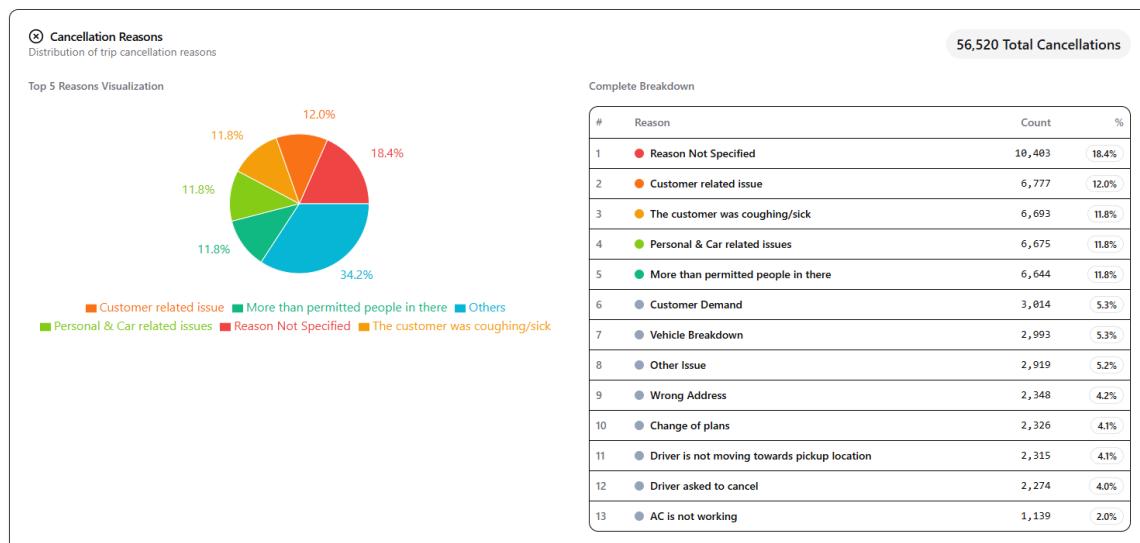
شکل ۳۵ کد مربوط به سفرهای لغو شده

در این کوئری:

COUNT(*) تعداد تکرار هر دلیل لغو را محاسبه می‌کند. سپس با استفاده از GROUP BY unified_cancellation_reason داده‌ها بر اساس دلیل لغو دسته‌بندی می‌شوند. و برای محاسبه درصد هر دلیل از کل لغوهای از Window Function استفاده شده است:

```
SUM(COUNT(*)) OVER () * 100, 2)
```

این عبارت مجموع کل لغوها را محاسبه می‌کند و سپس تعداد هر گروه بر آن تقسیم می‌شود تا درصد به دست آید. در نهایت نتایج بر اساس بیشترین تعداد مرتب شده‌اند تا مهم‌ترین دلایل در ابتدای نمودار نمایش داده شوند.



شکل ۳۶ نمودار دلایل لغو سفر

در فرانت‌اند نحوه نمایش حتی بهتر هم شده و در قسمت راست کلیه دلایل لغو سفرها رو بر اساس درصد نمایش داده‌ایم و در سمت چپ با نمودار دایره‌ای ابتدا ۵ نتیجه برتر (بیشترین دلایل لغو سفر) رو نمایش داده و بقیه رو (برای اینکه شلوغ نشود) در قسمت others قرار دادیم. این شکلی خیلی به تحلیل ما کمک بیشتری می‌کند. کمک می‌کند مشخص شود بیشترین علت شکست سفرها چه بوده و تمرکز بهبود سیستم باید روی کدام بخش باشد.

۲-۳-۲ - نمودار توزیع روش‌های پرداخت

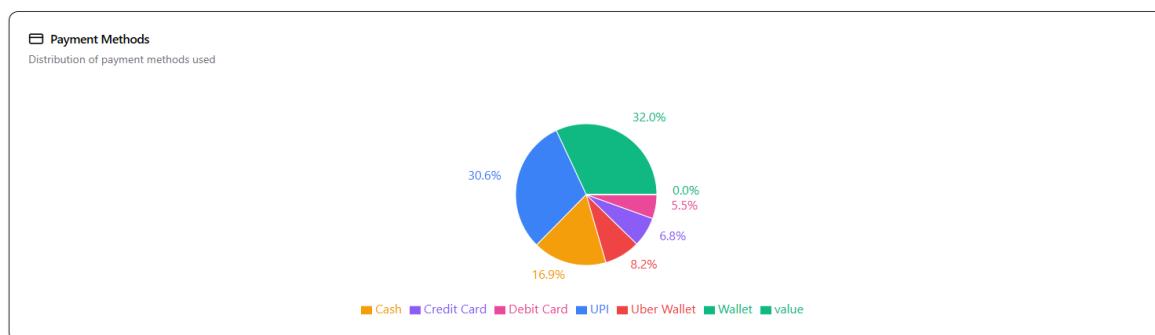
در این بخش، تحلیل روش‌های پرداخت انجام شده است. کوئری مربوط به تابع `getPaymentMethods` به صورت زیر است:

```
const query = `SELECT
    payment_method,
    COUNT(*) AS count,
    ROUND((COUNT(*)::NUMERIC / SUM(COUNT(*)) OVER ()) * 100, 2) AS percentage
FROM gold.gold_dataset
${whereSQL}
GROUP BY payment_method
ORDER BY count DESC
`;
```

در این کوئری:

داده‌ها بر اساس `payment_method` گروه‌بندی شده‌اند.

تعداد استفاده از هر روش پرداخت را محاسبه می‌کند، درصد هر روش نسبت به کل سفرها با همان منطق Window Function محاسبه شده است و نتایج بر اساس بیشترین استفاده مرتب شده‌اند.



شکل ۳۷ نمودار دایره‌ای روش‌های پرداخت

این نمودار نشان می‌دهد چه درصدی از کاربران از هر روش پرداخت استفاده کرده‌اند و برای تحلیل رفتار مالی بسیار مفید است، زیرا می‌توان تشخیص داد کدام روش محبوب‌تر است و آیا نیاز به توسعه یا بهبود روش خاصی وجود دارد یا خیر.

۲-۳-۳ - نمودار مقایسه نوع خودرو

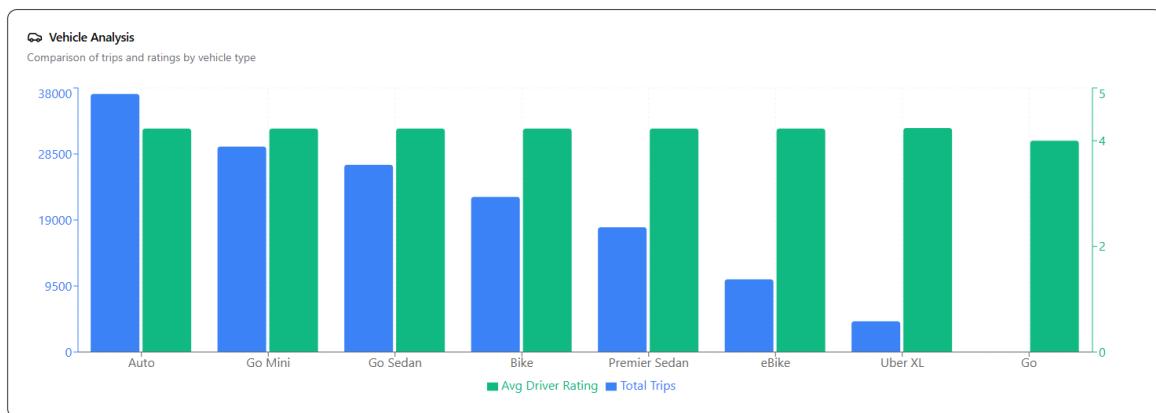
در این بخش، برای هر نوع خودرو موارد زیر محاسبه شدند:

- تعداد سفرهای موفق
- میانگین امتیاز راننده
- میانگین امتیاز مشتری
- مجموع درآمد تولیدشده
- میانگین ارزش هر رزرو

کوئری مربوط به تابع getVehicleAnalysis به صورت زیر است:

```
const query = `SELECT
    vehicle_type,
    COUNT(*) AS total_trips,
    ROUND(AVG(driver_rating), 2) AS avg_driver_rating,
    ROUND(AVG(customer_rating), 2) AS avg_customer_rating,
    SUM(booking_value) AS total_revenue,
    ROUND(AVG(booking_value), 2) AS avg_booking_value
FROM gold.gold_dataset
${whereSQL}
GROUP BY vehicle_type
ORDER BY total_trips DESC
`;
```

میانگین‌ها با استفاده از ROUND تا دو رقم اعشار گرد شده‌اند و داده‌ها بر اساس بیشترین تعداد سفر مرتب شده‌اند تا مشخص شود کدام نوع خودرو بیشترین استفاده را داشته است.



شکل ۳۸ نمودار میله‌ای مقایسه انواع خودرو

این بخش امکان تحلیل عملکرد هر کلاس خودرو را فراهم می‌کند و مشخص می‌کند کدام دسته بیشترین رضایت یا بیشترین استفاده را داشته است. همچنین می‌توان از آن برای تصمیم‌گیری در تخصیص منابع استفاده کرد.

۴-۳-۲- نمودار ساعات پر تردد

برای تحلیل الگوی زمانی استفاده از سیستم، از ستون زمان شروع سفر (request_datetime) استفاده شد. هدف این نمودار شناسایی ساعات پر تردد شباه روز است.

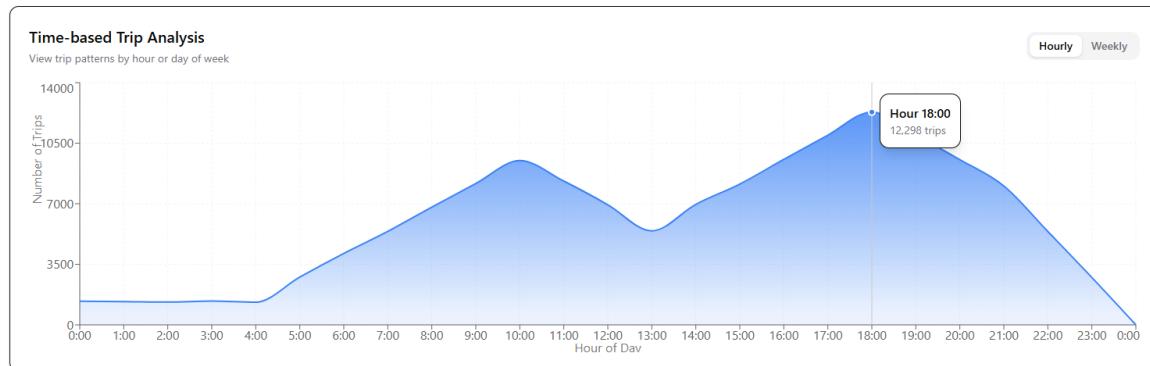
در این بخش، ساعت از فیلد تاریخ با استفاده از تابع EXTRACT(HOUR FROM ...) استخراج شده است. کوئری مربوط به تابع getPeakHours به صورت زیر است:

```
const query = `SELECT
    pickup_hour AS hour,
    COUNT(*) AS trip_count
FROM gold.gold_dataset
${whereSQL}
GROUP BY pickup_hour
ORDER BY pickup_hour`;
`;
```

در این کوئری:

با استفاده از EXTRACT(HOUR FROM request_datetime) ساعت هر سفر استخراج می‌شود سپس COUNT(*) ، تعداد سفرهای ثبت شده در هر ساعت را محاسبه می‌کند. و در نهایت داده‌ها بر

اساس ساعت به صورت صعودی مرتب می‌شوند تا نمودار خطی یا ستونی ترتیب طبیعی زمانی داشته باشد.



شکل ۳۹ نمودار خطی ساعت پرتردد

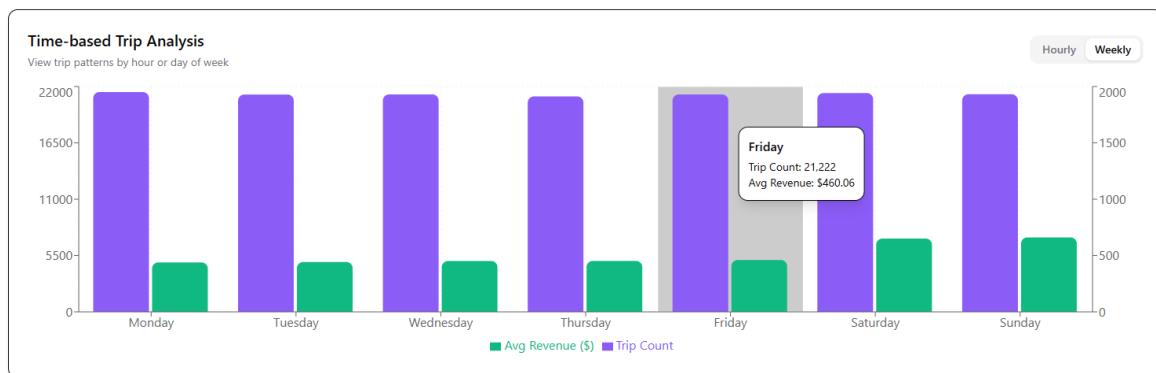
این نمودار نشان می‌دهد بیشترین حجم سفر در چه ساعاتی از شبانه‌روز اتفاق می‌افتد.

۵-۳-۲- نمودار تحلیل روزهای هفته

برای تحلیل رفتار کاربران در طول هفته، روز هفته از تاریخ سفر استخراج شده است

کوئری مربوط به تابع `getWeekdayAnalysis` به صورت زیر است:

```
const query = `SELECT
    day_name,
    COUNT(*) AS trip_count,
    ROUND(AVG(booking_value), 2) AS avg_revenue
FROM gold.gold_dataset
${whereSQL}
GROUP BY day_name
ORDER BY
    CASE day_name
        WHEN 'Monday' THEN 1
        WHEN 'Tuesday' THEN 2
        WHEN 'Wednesday' THEN 3
        WHEN 'Thursday' THEN 4
        WHEN 'Friday' THEN 5
        WHEN 'Saturday' THEN 6
        WHEN 'Sunday' THEN 7
    END`;
`;
```



شکل ۴۰ نمودار میله‌ای تعداد سفرها براساس روز هفته

همانطور که به صورت بصری هم دیده می‌شود سفرها تقریباً به اندازه یکسان در طول هفته انجام شده اند (البته این سفرها در بازه all-time یا کل داده‌ها) محاسبه شده‌اند. اما همانطور که دیده می‌شود به طور مشخص میانگین درآمد در روز‌های Saturday و Sunday یعنی شنبه - یکشنبه بیشتر بوده است. مانند بقیه نمودارها این نمودار نیز با هاور کردن روی هر قسمت اطلاعات کامل آن نمایش داده می‌شود.

گزارش فاز هفتم: دستیار هوشمند Text to SQL

خب در این بخش ما باید امکاناتی رو فراهم می‌کردیم که مدیر یا هر کاربری بتوانه به زبان ساده انگلیسی سوال بپرسه مثلًا What is the average rating by vehicle type? و سیستم خودش یک کوئری SQL درست و امن بسازه و نشون بده. مهم‌ترین چیز این بود که سیستم فقط SELECT بزن و هیچ وقت داده رو تغییر نده یا کوئری خطرناک ننویسه.

۲-۱- پیاده‌سازی بک‌اند

من اول یه فایل جداگانه به اسم routes توی sqlAssistant.js ساختم و یه اندپوینت POST به اسم /api/sql-assistant تعریف کدم. وقتی کاربر سوالش رو می‌فرسته، سیستم اول چک می‌کنه که سوال خالی نباشه و طولش بیشتر از ۵۰۰ کاراکتر نباشه (برای جلوگیری از درخواست‌های سنگین). بعد به پaramپت خیلی مهم به مدل می‌فرستم که شامل کل schema جدول gold_dataset هست. این رو توی یه ثابت به اسم GOLD_SCHEMA نگه داشتم که همه ستون‌ها، نوع داده‌ها و توضیحات مهم رو دقیق نوشته.

```

3 const GOLD_SCHEMA = ` 
4 Database Schema - Gold Layer (gold.gold_dataset):
5 Table: gold.gold_dataset
6 Columns:
7 - booking_id (TEXT, PRIMARY KEY): Unique booking identifier (e.g., 'CNR1234567')
8 - trip_timestamp (TIMESTAMP): Date and time of the trip
9 - pickup_hour (INTEGER): Hour of day (0-23)
10 - day_name (TEXT): Day of week name (e.g., 'Monday', 'Tuesday')
11 - is_weekend (BOOLEAN): Whether the trip occurred on weekend
12 - booking_status (TEXT): Status of booking (e.g., 'Completed', 'Cancelled')
13 - customer_id (TEXT): Customer identifier
14 - vehicle_type (TEXT): Type of vehicle (e.g., 'Sedan', 'Auto', 'eBike', 'Bike', 'Moto', 'Mini', 'XL', 'Go')
15 - unified_cancellation_reason (TEXT): Reason for cancellation if applicable
16 - booking_value (NUMERIC): Trip fare/revenue in dollars
17 - ride_distance (NUMERIC): Trip distance in kilometers
18 - revenue_per_km (NUMERIC): Revenue per kilometer
19 - driver_rating (NUMERIC): Driver rating (0-5)
20 - customer_rating (NUMERIC): Customer rating (0-5)
21 - payment_method (TEXT): Payment method used (e.g., 'Cash', 'Credit Card', 'Digital Wallet')
22 - driver_rating_imputed (BOOLEAN): Whether driver rating was imputed
23 - customer_rating_imputed (BOOLEAN): Whether customer rating was imputed
24 - is_cancelled (BOOLEAN): Whether the trip was cancelled
`
```

شکل ۴۱ توصیف GOLD_SCHEMA برای AI در کنترلر

پرامپت رو طوری نوشتم که مدل فقط SELECT تولید کنه و هیچ وقت کوئری مخرب مثل DELETE، UPDATE یا INSERT نسازه. اگر سوال خارج از حوزه داده‌های سفر بود (مثل احوالپرسی یا سوال برنامه‌نویسی)، مدل باید با پیام خطأ جواب بده.

همچنین اجبار کردم که حتماً LIMIT داشته باشه (حداقل ۱۰۰ اگر کاربر چیزی نگفته) تا کوئری سنگین نشه و سرور به مشکل نخوره.

```
// System prompt with all validation logic
const SYSTEM_PROMPT = `You are a SQL expert assistant for a ride-sharing analytics database. Your job is to convert natural language questions into valid SQL queries. You must follow these rules:
${GOLD_SCHEMA}

CRITICAL RULES YOU MUST FOLLOW:
1. ONLY generate SELECT queries. Absolutely refuse to generate INSERT, UPDATE, DELETE, DROP, TRUNCATE, ALTER, CREATE, etc.
2. If the user asks for data modification, respond with: "ERROR: Only SELECT queries are allowed. I cannot modify data."
3. If the question is not related to trip analytics (greetings, general questions, coding help, etc.), respond with: "I'm sorry, I can only handle trip analytics questions."
4. Always include a LIMIT clause. Default to LIMIT 100 if not specified.
5. Use proper PostgreSQL syntax and functions.
6. Always use the full table name: gold.gold_dataset
7. Return ONLY the SQL query as plain text. No explanations, no markdown formatting, no code blocks, no backticks.
8. Validate that all referenced columns exist in the schema above.
9. Use appropriate aggregations and GROUP BY clauses when needed.
10. For date filtering, use proper timestamp comparisons.
`
```

شکل ۴۲ تعریف محدودیت‌ها و اعتبارسنجی به صورت Prompt Engineering

برای ارتباط با مدل از OpenRouter استفاده کردم و مدل meta-llama/llama-3.3-70b-instruct را انتخاب کردم چون هم رایگان بود و هم برای تولید SQL خیلی خوب کار می‌کرد. توکن API را توی temperature را درخواست رو فرستادم. ۰.۱ env گذاشتم و با axios گذاشتم که خروجی دقیق و بدون خلاقيت اضافي باشه.

وقتی مدل کوئری رو برگردوند، اول چک کردم که با "ERROR: شروع نشده باشه. اگر خطأ بود، همون پیام رو به کاربر نشون دادم. اگر کوئری معتبر بود، مستقیم به فرانتاند برگردوندم تا توی چت نمایش داده بشه.

در فرانتاند هم یه صفحه ساده با Tailwind و Shadcn ساختم. وقتی کاربر سوال می‌پرسه، اول پیام خودش رو توی چت نشون می‌ده (اینجا خیلی دقیق که پیام کاربر حتماً و فوری نمایش داده بشه)، بعد درخواست رو به بکاند می‌فرستم. وقتی جواب برگشت، پیام assistant رو با متن "Here's the SQL query for your question: " اضافه می‌کنم و کد SQL رو توی یه بلوك جداگانه با SyntaxHighlighter نمایش می‌دهم:

برای نمایش کد SQL هم از oneLight react-syntax-highlighter استفاده کردم که روشن و خوانایت دارد. در نهایت یه دکمه کپی اضافه کردم که با کلیک، کوئری رو توی کلیپبورد می‌ذاره و تیک سبز نشون می‌ده. همچنین یه بخش کوچیک کنار چت گذاشتم که schema جدول رو خلاصه نشون بده تا کاربر بدونه چه ستون‌هایی داره.

شكل ۴۳ دستیار هوشمند Text to SQL

The screenshot shows the SQL Assistant interface. On the left, under 'Chat with Assistant', a user asks: 'Show the average revenue per vehicle type, along with number of trips, total distance, and driver points earned for each month through 2024, for completions only (non-cancellations). Results are sorted by month and vehicle type.' Below this, the generated SQL query is displayed:

```

SELECT
    EXTRACT(
        MONTH
        FROM
            trip_timestamp
    ) AS trip_month,
    EXTRACT(
        YEAR
        FROM
            trip_timestamp
    ) AS trip_year,
    vehicle_type,
    AVG(booking_value) AS avg_revenue,
    COUNT(*) AS num_trips,
    SUM(ride_distance) AS total_distance,
    SUM(driver_rating) AS total_driver_rating
FROM
    gold.gold_dataset
WHERE
    is_cancelled = FALSE
    AND trip_timestamp < '2025-01-01'
GROUP BY
    EXTRACT(
        MONTH
        FROM
            trip_timestamp
    ),
    EXTRACT(
        YEAR
    )

```

On the right, under 'Generated SQL', the query is shown again with a 'Copy' button. Below it, under 'Example Questions', several sample queries are listed:

- Show me the top 10 highest revenue trips
- What is the average rating by vehicle type?
- How many trips were cancelled by each reason?
- Show completed trips with revenue above \$50
- What are the peak hours for rides?
- Compare total trips by payment method
- Show average distance by vehicle type

شکل ۴۴ سوال پیچیده در دستیار هوشمند

برای تست، چند سوال مختلف زدم و دیدم سیستم خیلی خوب کار می‌کنه.

The screenshot shows the SQL Assistant interface. In the 'Chat with Assistant' section, a user asks: 'Show me the top 10 customers by total spending...'. A response from the assistant says: 'Hi How are you?'. Below the input field, there is a red border around the message area with the text: 'Please ask questions about trip analytics data only.' and 'Please ask questions about trip analytics data only.' at the bottom.

In the 'Example Questions' section on the right, the same sample queries are listed as in the previous screenshot.

شکل ۴۵ سوال نامربوط در دستیار هوشمند

اگر سوال نامربوط می‌پرسیدیم هم طبق تعریف اولیه باید خطأ می‌گرفتیم.

گزارش فاز هشتم: بهینه‌سازی و ایندکس‌گذاری

در این مرحله تأثیر ایندکس‌گذاری را روی یک کوئری سنگین رو بررسی می‌کنیم. مراحل انجام شده و نتایج به دست آمده در ادامه ارائه می‌شود.

۱-۲- انتخاب کوئری سنگین

هدف از این تست بررسی تأثیر ایندکس پوششی (Covering Index) بر بهینه‌سازی کوئری‌های سنگین و کاهش زمان اجرا بود. در این آزمایش، سعی شد کوئری‌ای انتخاب شود که:

- فیلترهای ترکیبی روی ستون‌های مهم داشته باشد مانند: `trip_timestamp, booking_value`
- گروه‌بندی و جمع‌آوری داده‌ها انجام دهد (`GROUP BY ,SUM, AVG, COUNT`)
- روی حجم داده واقعی جدول تست شود تا اثر index کاملاً مشهود باشد.

```
SELECT
    vehicle_type,
    payment_method,
    COUNT(*) AS total_trips,
    SUM(booking_value) AS total_revenue,
    AVG(driver_rating) AS avg_driver_rating
FROM gold.gold_dataset
WHERE
    booking_status = 'Completed'
    AND vehicle_type = 'Uber XL'
    AND trip_timestamp >= '2024-03-01'
    AND trip_timestamp < '2024-03-10'
GROUP BY vehicle_type, payment_method
ORDER BY total_revenue DESC;
```

دلایل انتخاب این کوئری این بود که:

- فیلتر روی چند ستون که ایندکس بتواند روی آن‌ها عمل کند
- گروه‌بندی روی ستون‌های کلیدی (vehicle_type, payment_method) دارد.
- محاسبه توابع تجمعی (COUNT, SUM, AVG) که روی داده‌های زیاد زمان بر است.
- محدوده زمانی مشخص تا بتوان تأثیر index را به وضوح دید.

۲-۲- اجرای کوئری قبل از ایندکس‌گذاری

با استفاده از EXPLAIN ANALYZE کوئری را اجرا کرده و خروجی زیر را بدست آوردهیم:

```
^
uber_db=# EXPLAIN ANALYZE
uber_db# SELECT
uber_db#     vehicle_type,
uber_db#     payment_method,
uber_db#     COUNT(*) AS total_trips,
uber_db#     SUM(booking_value) AS total_revenue,
uber_db#     AVG(driver_rating) AS avg_driver_rating
uber_db# FROM gold.gold_dataset
uber_db# WHERE
uber_db#     booking_status = 'Completed'
uber_db#     AND vehicle_type = 'Uber XL'
uber_db#     AND trip_timestamp >= '2024-03-01'
uber_db#     AND trip_timestamp < '2024-03-10'
uber_db# GROUP BY vehicle_type, payment_method
uber_db# ORDER BY total_revenue DESC;
```

شکل ۴۶ اجرای کوئری سنجین بدون ایندکس‌گذاری

```

uber_db=# SET enable_indexscan = off;
SET
uber_db=# EXPLAIN ANALYZE
uber_db# SELECT
uber_db#   vehicle_type,
uber_db#   payment_method,
uber_db#   COUNT(*) AS total_trips,
uber_db#   SUM(booking_value) AS total_revenue,
uber_db#   AVG(driver_rating) AS avg_driver_rating
uber_db# FROM gold.gold_dataset
uber_db# WHERE
uber_db#   booking_status = 'Completed'
uber_db#   AND vehicle_type = 'Uber XL'
uber_db#   AND trip_timestamp >= '2024-03-01'
uber_db#   AND trip_timestamp < '2024-03-10'
uber_db# GROUP BY vehicle_type, payment_method
uber_db# ORDER BY total_revenue DESC;
                                         QUERY PLAN
-----



Sort  (cost=236.87..236.89 rows=5 width=85) (actual time=1.957..1.958 rows=5.00 loops=1)
  Sort Key: (sum(booking_value)) DESC
  Sort Method: quicksort  Memory: 25kB
  Buffers: shared hit=70
    -> GroupAggregate (cost=235.94..236.81 rows=5 width=85) (actual time=1.930..1.949 rows=5.00 loops=1)
        Group Key: payment_method
        Buffers: shared hit=70
          -> Sort  (cost=235.94..236.10 rows=64 width=24) (actual time=1.916..1.919 rows=65.00 loops=1)
              Sort Key: payment_method
              Sort Method: quicksort  Memory: 27kB
              Buffers: shared hit=70
                -> Bitmap Heap Scan on gold_dataset  (cost=5.40..234.02 rows=64 width=24) (actual time=0.142..1.817 rows=65.00 loops=1)
                    Recheck Cond: ((booking_status = 'Completed'::text) AND (vehicle_type = 'Uber XL'::text) AND (trip_timestamp >= '2024-03-01 00:00:00'::timestamp without time zone))
                    Heap Blocks: exact=63
                    Buffers: shared hit=67
                      -> Bitmap Index Scan on idx_covering_force  (cost=0.00..5.38 rows=64 width=0) (actual time=0.050..0.050 rows=65.00 loops=1)
                          Index Cond: ((booking_status = 'Completed'::text) AND (vehicle_type = 'Uber XL'::text) AND (trip_timestamp >= '2024-03-01 00:00:00'::timestamp without time zone))
                          Index Searches: 1
                          Buffers: shared hit=4
Planning Time: 0.224 ms
Execution Time: 2.002 ms
(21 rows)

```

شکل ۴۷ خروجی کوئری سنتگین بدون ایندکس‌گذاری

Planning Time: 0.223 ms

Execution Time: 2.002 ms

تحلیل خروجی قبل از ایندکس:

اجرای کوئری بدون استفاده از ایندکس (Index Scan) غیرفعال شد

در این مرحله با دستور:

```
SET enable_seqscan = OFF;
```

PostgreSQL مجبور شد که حتی با وجود ایندکس موجود، از آن استفاده نکند و کوئری را با یک

مسیر EXPLAIN ANALYZE کامل اجرا کند. خروجی Bitmap Heap Scan یا Seq Scan نشان

می‌دهد:

- نوع اسکن Bitmap Heap Scan: روی جدول gold_dataset
- زمان اجرای واقعی: 2.002 ms
- تعداد ردیفها: 65

- استفاده از بافرها (shared hit=70 read=0): معمولاً نشان می‌دهد همه داده‌ها از کش یا حداقل گرفته شده‌اند (O/I).

تحلیل:

با غیرفعال کردن Index Scan، سیستم مجبور شد همه بلوک‌های مرتبط با شرط WHERE را در جدول اسکن کند، حتی اگر ایندکس وجود داشت. زمان اجرای کوئری به نسبت استفاده از ایندکس تقریباً ۱۵ برابر بیشتر شد (با فرض مشاهده زمان واقعی اجرای کوئری با ایندکس بعداً).

این حالت نشان می‌دهد که استفاده از ایندکس چقدر می‌تواند روی سرعت کوئری تاثیرگذار باشد، مخصوصاً روی جدول‌های بزرگ با فیلترهای ترکیبی.

۲-۳- ایجاد ایندکس‌های مناسب

با توجه به فیلترهای کوئری (بازه زمانی، برابری وضعیت و متد پرداخت) و همچنین نیاز به ستون‌های aggregation، تصمیم به ایجاد ایندکس ترکیبی پوششی (Covering Index) گرفته شد. در PostgreSQL از ایندکس B-Tree پیش‌فرض استفاده می‌کنیم که برای محدوده (range) و برابری (equality) مناسب است.

ایндکس انتخابی ما:

```
CREATE INDEX idx_covering_force
ON gold.gold_dataset
(bookings_status, vehicle_type, trip_timestamp)
INCLUDE (booking_value, payment_method, driver_rating);
```

اما چرا این ایندکس رو انتخاب کردیم؟

- ستون‌های booking_status، vehicle_type، trip_timestamp در کلید اصلی ایندکس قرار گرفته‌اند تا فیلتر سریع باشد.
- ستون‌های booking_value، payment_method، driver_rating در بخش INCLUDE قرار گرفته‌اند تا ایندکس PostgreSQL یا به صورت پوششی شود و PostgreSQL بتواند Index Only Scan انجام دهد بدون نیاز به مراجعه زیاد به Heap.

- نوع ایندکس B-Tree (معادل B+Tree) برای این نوع کوئری بهینه است.

۲-۴- خروجی کوئری با استفاده از ایندکس

گاهی اوقات اگر بخواهیم استفاده از ایندکس را به عهده Planer بگذاریم دیتابیس از آن استفاده نمی‌کند اما چرا؟

درصد بالایی از جدول فیلتر match می‌شود و وقتی درصد زیادی از جدول باید خوانده شود (مثلاً ۴۰٪ یا ۵۰٪)، استفاده از index سودی ندارد چون بعدش باید تقریباً همه heap page ها را بخواند.

همچنین جدول ما کوچک است و ۱۵۰۰۰ رکورد دارد برای همین PostgreSQL ترجیح میده Parallel Seq Scan بزن. اما برای اینکه نشون بدمی اندیس‌گذاری تاثیر بسیار زیادی می‌تواند روی حجم بالایی از داده‌ها بگذارد ما باز تنظیم کردیم تا ببینیم آیا Planner استفاده می‌کند یا خیر؟

```
SET enable_seqscan = ON;
```

```
uber_db=# SET enable_indexscan = on;
SET
uber_db=# EXPLAIN ANALYZE
uber_db# SELECT
uber_db#   vehicle_type,
uber_db#   payment_method,
uber_db#   COUNT(*) AS total_trips,
uber_db#   SUM(booking_value) AS total_revenue,
uber_db#   AVG(driver_rating) AS avg_driver_rating
uber_db# FROM gold.gold_dataset
uber_db# WHERE
uber_db#   booking_status = 'Completed'
uber_db#   AND vehicle_type = 'Uber XL'
uber_db#   AND trip_timestamp >= '2024-03-01'
uber_db#   AND trip_timestamp < '2024-03-10'
uber_db# GROUP BY vehicle_type, payment_method
uber_db# ORDER BY total_revenue DESC;
                                         QUERY PLAN
-----
```

```
Sort  (cost=10.79..10.81 rows=5 width=85) (actual time=0.097..0.098 rows=5.00 loops=1)
  Sort Key: (sum(booking_value)) DESC
  Sort Method: quicksort  Memory: 25kB
  Buffers: shared hit=5
    -> HashAggregate  (cost=10.66..10.74 rows=5 width=85) (actual time=0.085..0.089 rows=5.00 loops=1)
        Group Key: payment_method
        Batches: 1  Memory Usage: 32kB
        Buffers: shared hit=5
          -> Index Only Scan using idx_covering_force on gold_dataset  (cost=0.42..10.02 rows=64 width=24) (actual time=0.033..0.049 rows=65.00 loops=1)
              Index Cond: ((booking_status = 'Completed'::text) AND (vehicle_type = 'Uber XL'::text) AND (trip_timestamp >= '2024-03-01 00:00:00'::timestamp without time zone) AND (trip_timestamp < '2024-03-10 00:00:00'::timestamp without time zone))
                  Heap Fetches: 0
                  Index Searches: 1
                  Buffers: shared hit=5
Planning Time: 0.224 ms
Execution Time: 0.138 ms
(15 rows)
```

شکل ۴۸ خروجی کوئری سنتیکین با استفاده از ایندکس ما و اجرای همان کوئری، خروجی نشان داد:

- نوع اسکن idx_covering_force: روی ایندکس Index Only Scan
- زمان اجرای واقعی : 0.138 ms

- تعداد ردیفها: 65

▪ این یعنی به صورت کامل اطلاعات مورد نیاز را از Index Covering-Heap Fetches: 0 ایندکس خوانده است. همچنین استفاده از بافرها shared hit=5 است.

تحلیل:

با فعال بودن ایندکس، PostgreSQL توانست تمام داده‌های مورد نیاز برای فیلترها و حتی ستون‌های SELECT را از ایندکس پوششی (INCLUDE) بخواند و نیازی به اسکن جدول نداشت.

زمان اجرا به شدت کاهش یافته است، که نشان‌دهنده افزایش کارایی بیش از ۱۴ برابر فقط با استفاده از ایندکس است.

Index Only Scan باعث می‌شود که هیچ دسترسی اضافی به جدول (Heap) انجام نشود و فقط با خواندن ایندکس، محاسبات گروه‌بندی و جمع‌آوری داده‌ها انجام شود.

و این نشان‌دهنده اهمیت طراحی ایندکس‌های پوششی (Covering Index) برای کوئری‌های تحلیلی با فیلترهای متعدد و ستون‌های aggregate است.

۲-۵ - مقایسه

جدول ۳ مقایسه استفاده از ایندکس و عدم استفاده از ایندکس مناسب

ویژگی/حالت	<i>Sq Scan (Index Off)</i>	<i>Index Only Scan (Index On)</i>
زمان واقعی اجرا	2.002 ms	0.138 ms
نوع اسکن	Bitmap Heap Scan	Index Only Scan
Heap Fetches	yes	0
تعداد ردیفها	65	65
استفاده از بافرها	70	5
تحلیل عملکرد	کندتر و پرهزینه‌تر	بسیار سریع و بهینه

گزارش فاز نهم : جست و جوی معنای (Semantic Search) و Vector Database

این بخش اختیاری است.

منابع و مراجع

کتاب‌ها و منابع آموزشی اصلی

- PostgreSQL Documentation (نسخه ۱۶ و بالاتر)، وبسایت رسمی PostgreSQL <https://www.postgresql.org/docs/> دسترسی در بهمن ۱۴۰۳
- Medallion Architecture (Data Lakehouse) ، مقاله و راهنمای Databricks در بهمن ۱۴۰۳ <https://www.databricks.com/glossary/medallion-architecture>
- Designing Data-Intensive Applications (فصل‌های مرتبط با ETL ، ایندکس‌گذاری و معماری داده) O'Reilly Media .Martin Kleppmann .

ابزارها و تکنولوژی‌های استفاده شده

- Express.js و Node.js ، وبسایت رسمی، دسترسی در بهمن ۱۴۰۳ <https://expressjs.com>
- React.js با Tailwind CSS و Shadcn/ui ، وبسایت‌های رسمی، دسترسی در بهمن ۱۴۰۳ <https://ui.shadcn.com> <https://tailwindcss.com> <https://react.dev>
- PostgreSQL برای تمام لایه‌های Gold و Silver ، Bronze و WebSQL ، وبسایت رسمی، دسترسی در بهمن ۱۴۰۳
- OpenRouter (برای مدل‌های رایگان LLM در Text-to-SQL)، وبسایت رسمی، دسترسی در بهمن ۱۴۰۳ <https://openrouter.ai>
- Chroma DB برای Vector Database و جستجوی معنایی، وبسایت رسمی، دسترسی در بهمن ۱۴۰۳ <https://www.trychroma.com>
- GitHub برای نمایش زیبای کد SQL در چت، GitHub ریپازیتوری، React-Syntax-Highlighter برای نمایش زیبای کد SQL در چت، GitHub دسترسی در بهمن ۱۴۰۳ <https://github.com/react-syntax-highlighter/react-syntax-highlighter>

منابع هوش مصنوعی و مدل‌ها

- Llama 3.3 70B Instruct مدل اصلی استفاده شده در دستیار Text-to-SQL
- Prompt Engineering برای Text-to-SQL ، مقاله‌های Hugging Face و راهنمای OpenAI در بهمن ۱۴۰۳ <https://huggingface.co/docs/transformers/tasks/text-to-sql>

منابع تكميلی و الهام‌بخش

- revenue برای درک بهتر ساختار داده‌های سفر و متریک‌ها مثل Uber Engineering Blog <https://www.uber.com/blog/engineering/>، در بهمن ۱۴۰۳ per km دسترسی
- راهنمای ایندکس‌گذاری و بهینه‌سازی کوئری در PostgreSQL ، و بلاگ Percona و Citus
- Data، در بهمن ۱۴۰۳ دسترسی در بهمن ۱۴۰۳ مدالی Medallion Architecture در پروژه‌های واقعی، و بلاگ dbt Labs و
- Snowflake، در بهمن ۱۴۰۳ دسترسی در بهمن ۱۴۰۳ ابزارهای توسعه و تست
- Postman برای تست کامل API‌های CRUD و Text-to-SQL، وب‌سایت رسمی، در بهمن ۱۴۰۳ <https://www.postman.com>
- pgAdmin برای اجرای کوئری‌های EXPLAIN ANALYZE و بررسی پلن اجرایی در بهمن ۱۴۰۳
- Vite.js برای اجرا کدهای سمت فرانت اند

پیوست‌ها:

لینک ریپازیتوری:

<https://github.com/HosseinMst81/uber-data-platform/>

