

NYC Taxi Trip Explorer - Technical Report

Executive Summary

So we had to build this NYC Taxi Trip Explorer for our class - basically a web app that digs through taxi trip data and shows cool patterns. This report is us explaining how we actually built the thing, what broke (a lot), and what we figured out.

1. Project Setup & Design

1.1 Three-Layer Setup

We went simple with three parts:

Frontend - Plain HTML/CSS/JavaScript with Chart.js for charts. We almost used React but honestly, that seemed like way too much work for what we needed. Sure, we had to write more code to mess with the DOM, but at least we knew what was going on.

Backend - Flask REST API. We broke it into separate files so we wouldn't lose our minds looking for stuff. Flask was perfect because it's simple and we all already knew how to use it. Django would've been like using a bulldozer to plant flowers.

Database - SQLite because it just works. Yeah, it can't handle a million users, but for our demo it's great. Plus it comes with everything we need already built in.

1.2 Database Setup

We tried to do what our professor showed us about organizing data:

```
-- zones table: all the NYC taxi areas
-- trips table: the actual rides
```

What we did:

1. Put zones and trips in different tables - no point copying "Manhattan" a million times
2. Added indexes on stuff we'd be searching through a lot
3. Did all the calculations (trip time, speed, rush hour) when loading data, not every single query
4. Added rules so junk data gets thrown out automatically

Doing the math ahead of time takes more space but makes everything way faster. Totally worth it.

1.3 How Our Tables Connect

Here's our database layout:

```
+-----+
|                ZONES                |
+-----+
| PK LocationID (INT)                  |
|    Borough (TEXT)                   |
+-----+
```

Zone (TEXT)	
service_zone (TEXT)	
+-----+	
	1:M
+-----+	
TRIPS	
+-----+	
PK trip_id (INT AUTO)	
pickup_datetime	
dropoff_datetime	
passenger_count	
trip_distance	
FK PULocationID	
FK DOLocationID	
payment_type	
fare_amount	
total_amount	
duration_minutes	
speed_mph	
is_rush_hour	
pickup_hour	
... (other stuff)	
+-----+	

Each zone can have tons of trips starting or ending there. We made sure garbage data gets kicked out and added indexes so searches don't take forever.

2. Our Sorting Algorithm

2.1 Why We Made Our Own QuickSort

We decided to write our own sorting instead of using Python's. Sounds stupid but here's why:

```
def quicksort(arr, key_func=None):
    # Split and sort recursively
```

Why QuickSort: Fast enough for our data ($O(n \log n)$ usually), doesn't eat memory like merge sort, and we can sort by any field with that `key_func` thing.

We looked at merge sort but it uses more memory and we didn't need it to keep equal items in order. The `key_func` was smart - lets us sort by fare, distance, whatever, without writing different functions.

2.2 Speed Test

How fast it runs (50 trips): Sorting by fare takes 0.002ms, same for distance and time.

For real apps with millions of records, we'd just let the database handle sorting and use pages so we don't crash browsers.

3. Data Processing

3.1 Cleaning Up the Data

We made a simple process:

Extract: Download NYC taxi CSV files

Transform: Toss garbage data (negative distances, zero fares), calculate trip time and speed, figure out rush hour, fix data types

Load: Dump everything into SQLite in chunks

We do the heavy work when loading instead of every query. Takes longer to load but way faster when people use it.

3.2 Data Problems

Cleaning data was horrible:

1. **Database rules:** Catch obviously wrong stuff
2. **Python filtering:** Pandas removes weird outliers
3. **Rush hour logic:** Real NYC traffic (7-9 AM, 5-7 PM)

The taxi data is nuts. We found trips with zero distance but \$50 fares, negative trip times, pickups after dropoffs. Took forever to figure out what to keep. Way harder than expected.

4. API Setup

4.1 Simple Endpoints

We made five endpoints:

1. `/api/trips` - Get trip data
2. `/api/zones` - Get zone info
3. `/api/analytics/top-routes` - Popular routes
4. `/api/analytics/rush-hour` - Traffic by hour
5. `/api/sort-trips` - Show our sorting

We could've crammed everything into one but separate ones are easier to debug. Added limits so we don't send 100,000 records and kill browsers.

4.2 CORS Fix

Had to enable CORS so frontend could talk to backend:

```
CORS(app) # Let any site connect - would fix for real use
```

Real apps would lock this down with authentication.

5. Frontend Stuff

5.1 What We Used

Chart.js for graphs because it's simple and looks okay. We checked D3.js but way too complicated for basic charts.

UI choices: Card layout looks clean on phones, dropdown for trying different sorts, async/await so page doesn't freeze.

5.2 Speed Tricks

Making it fast: Show limited records so page doesn't bog down, load data in background, handle network errors.

Whole page loads in under half a second with our test data.

6. Problems We Hit

6.1 Big Issues

Messy data: NYC changes file formats every year, field names and types all different. Built flexible processor for various formats. Real data way messier than textbooks.

Algorithm stuff: Had to figure out sorting through web API efficiently. The key_func saved us - one algorithm, multiple sorts.

Slow database: Complex JOIN queries taking forever. Added indexes and optimized. Now everything under 100ms.

6.2 Team Stuff

Four people was harder than expected. Everyone stepping on each other's code at first. Then got organized: defined APIs upfront, clear ownership (backend, frontend, data, database), used Git right (after deleting frontend twice).

Once we knew who owned what, much smoother.

7. Testing & Speed

Our QuickSort only 5% slower than Python's. Not bad for students!

Database times: Top routes: 45ms, Rush hour: 32ms, Trip sorting: 15ms

What we tested: QuickSort with different sizes, database rules, API endpoints, browser stuff.

8. What We Figured Out

Algorithms: Writing QuickSort was cool but Python's is probably better for real stuff. Sometimes you gotta build it to get it.

Data: Cleaning real data takes way longer than you think. Next time budget more time just for cleanup.

Teamwork: Clear boundaries between code saved us. Made debugging easier when you knew whose code broke.

Planning: Agreeing on API early prevented arguments. Clear ownership was key.

Tech: Simple Flask and vanilla JavaScript was right. Got stuff done instead of fighting frameworks.

9. If We Had More Time

Tech stuff: Live data streaming, machine learning predictions, interactive maps, better database for multiple users, caching.

User stuff: User accounts, advanced filtering, export reports, mobile design.

10. Wrap Up

We built a working NYC taxi analyzer. Main wins: Clean setup that let us work in parallel, custom sorting that works, solid pipeline for messy real data, interactive web interface, decent documentation.

Main lesson: Keep it simple on deadlines. Could've used fancy frameworks but basic approach worked better.

Good learning experience and showed we can work together to build useful stuff.

Who Did What:

- Dianah Shimwa Gasasira: Backend API development and custom algorithm implementation
- Ayobamidele Aiyedogbon: Frontend interface and data visualization components
- Jesse Nkubito: Data processing pipeline and comprehensive documentation
- Bior Aguer Kuir: Database design and schema optimization