JOB AGGREGATOR LITE

PROJECT 8

Jack Kammerer

CS.4332.252 Introduction to Database Systems

Texas State University

April 2025

Contents

1 Introduction

At this milestone, I successfully finalized the backend functionality of my LinkedIn Job Scraper project. This phase required major pivots, refactoring, and plenty of trial and error to achieve a stable, fully functional scraper backed by a normalized database. The project originally began with a minimal design: using SQLite and just two basic entities. But to meet the expanded project scope and ensure proper database normalization, I scaled the system into a five-entity architecture, now running on MySQL and fully integrated with **phpMyAdmin** for easier data management and visibility. Early on, I explored using AI-powered parsing tools like **Ollama** and **CodeLlama** hosted locally, in an attempt to automate SQL generation and parsing tasks. However, due to their limited model sizes and inefficiencies with complex queries, I pivoted to building out custom, hard-coded parsing logic. This allowed me to reliably extract and structure job data aligned with the roles and industries I personally care about.

As of Project 8, my system achieves the following:

- Fully operational LinkedIn scraper with a hardcoded URL targeting relevant job listings
- Clean and efficient parsing logic for job titles, company names, locations, skills, and post dates
- Expansion from 2 to 5 entities, achieving a normalized database schema (3NF)
- Data storage in MySQL with full visibility and control via phpMyAdmin
- Real-time data scraping and insertion, verified through terminal outputs and phpMyAdmin dashboards

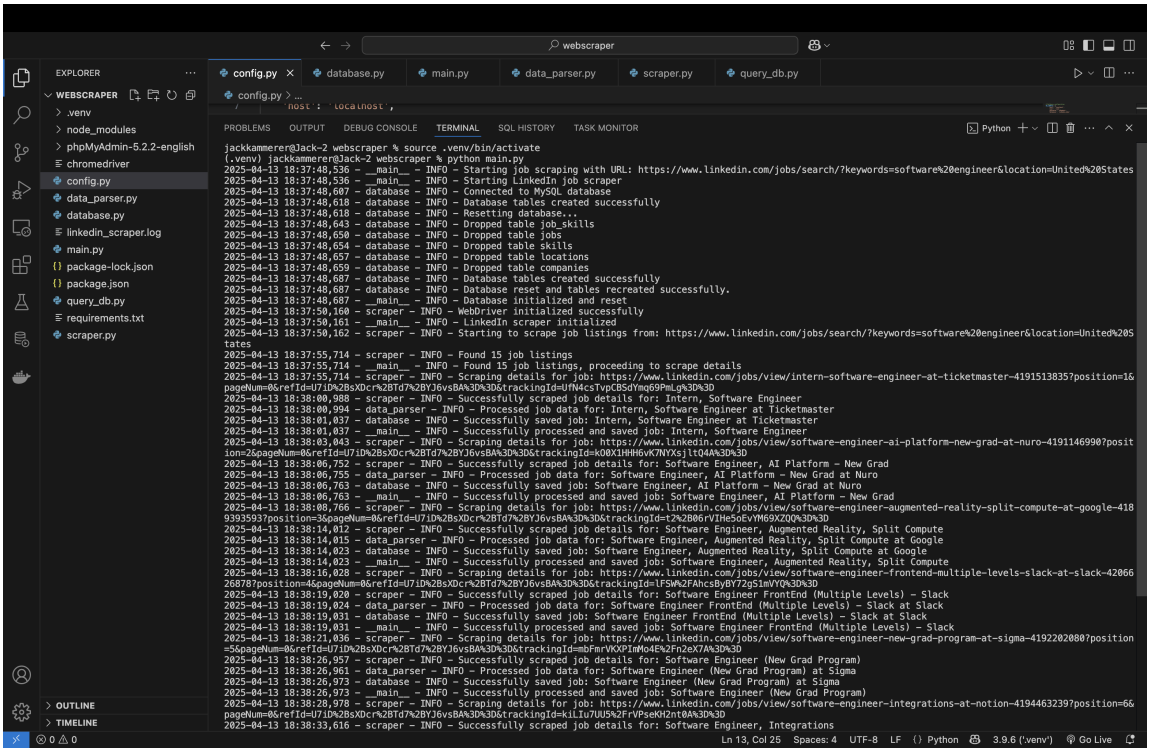

Figure 1: main.py executing(part1)

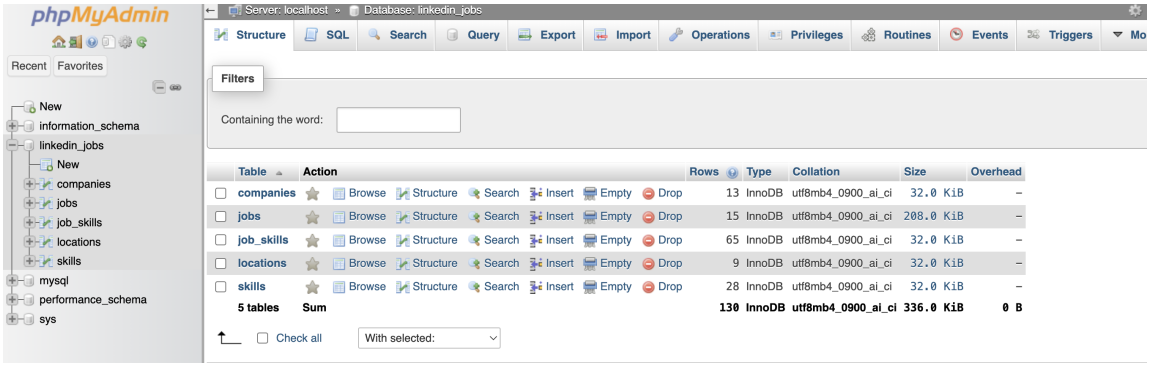

Figure 2: Data Scraped Tables

2 Updated E/R Diagram

The E/R diagram evolved substantially from the earlier phase. Originally designed with just `job_sites` and `job_postings`, I expanded the model to five entities:

- **Jobs**: Job title, description, post date, etc.

- **Companies**: Company name, industry, size
- **Locations**: City, state, country
- **Skills**: Technologies or skills required
- **Job_Skills**: Many-to-many relationship table between jobs and skills

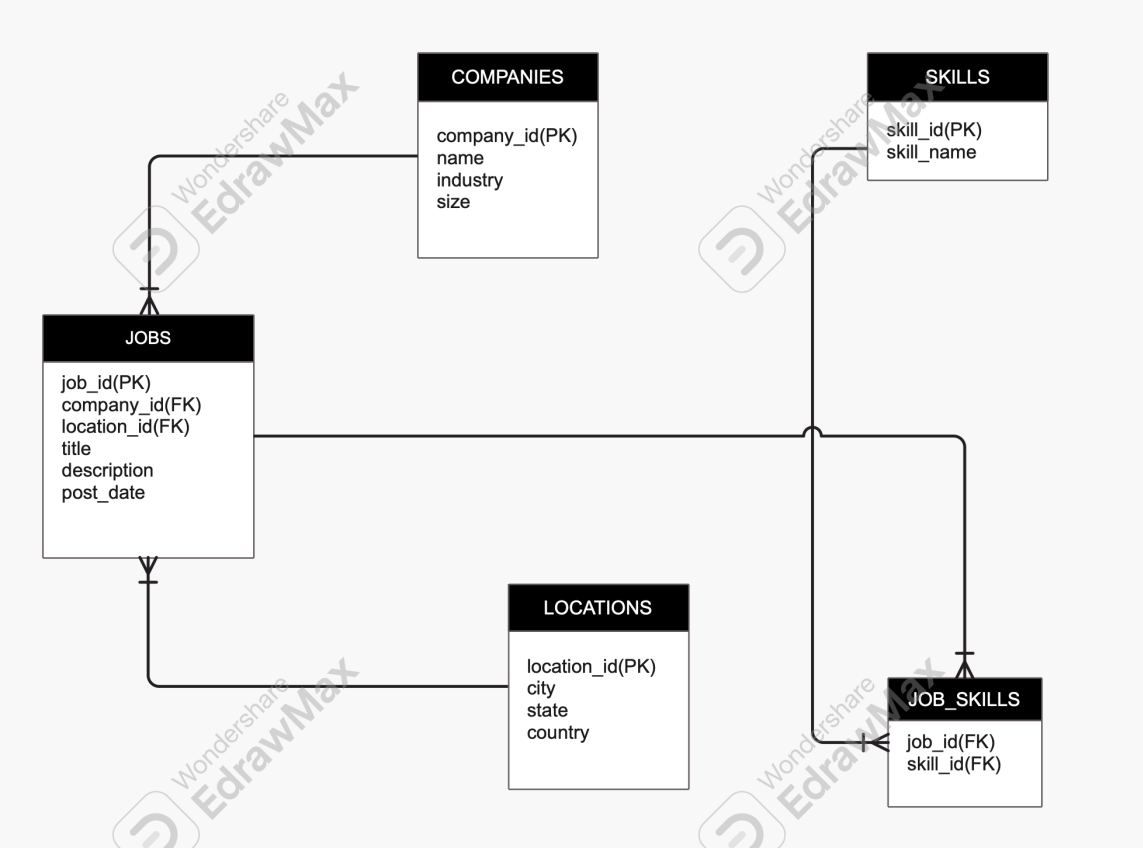

Figure 3: New E/R Diagram - 3NF

3 English Queries and SQL Implementations

Due to major changes in my project to fit the expansion standard (migrating from a 2-entity model to a 5-entity normalized database), my original English queries were revisited and adjusted to suit the final database design.

3.1 Query 1: Tech Job Overview

English:

List all tech job postings scraped from LinkedIn, showing the job title, company name, location (city, state, country), and the post date. **SQL:**

```
SELECT j.title, c.name AS company_name, l.city, l.state, l.country, j.post_date
FROM jobs j
JOIN companies c ON j.company_id = c.company_id
JOIN locations l ON j.location_id = l.location_id
ORDER BY j.post_date DESC;
```

Screenshot:

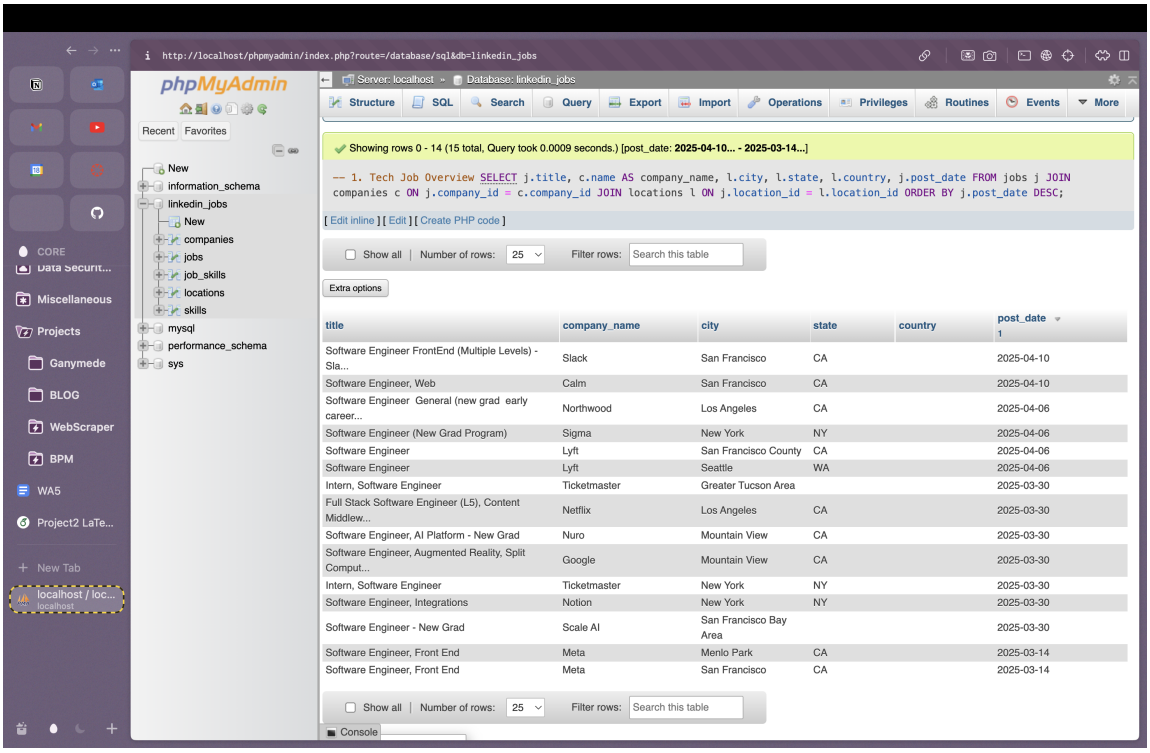

Figure 4: Query 1

3.2 Query 2: Web Development Focus

English:

Retrieve all job postings where the job title or description contains 'web development', and sort them by the most recent post date. **SQL:**

```
SELECT j.title, c.name AS company_name, l.city, l.state, l.country, j.post_date
FROM jobs j
JOIN companies c ON j.company_id = c.company_id
JOIN locations l ON j.location_id = l.location_id
WHERE j.title LIKE '%web development%' OR j.description LIKE '%web development%'
ORDER BY j.post_date DESC;
```

Screenshot:

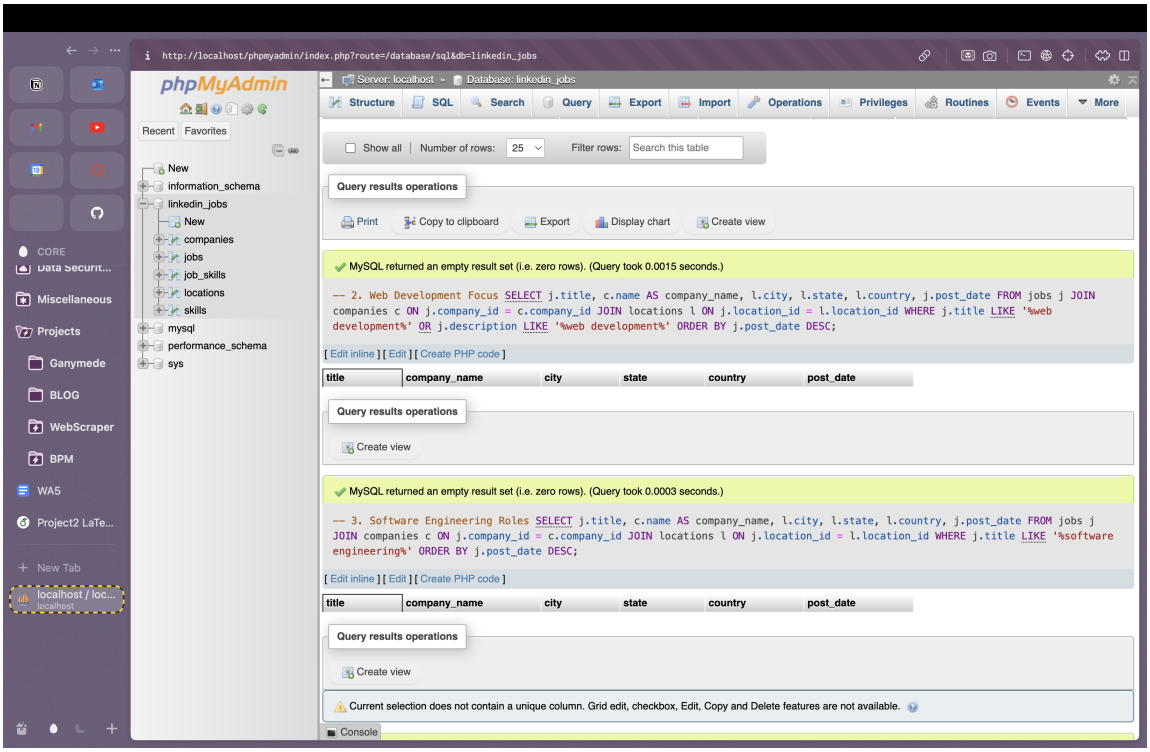

Figure 5: Query 2

3.3 Query 3: Software Engineering Roles

English:
Find job postings with 'software engineering' in the title and display the associated company name, full location, and post date. **SQL:**

```
SELECT j.title, c.name AS company_name, l.city, l.state, l.country, j.post_date
FROM jobs j
JOIN companies c ON j.company_id = c.company_id
```

```
JOIN locations l ON j.location_id = l.location_id
WHERE j.title LIKE '%software engineering%'
ORDER BY j.post_date DESC;
```

Screenshot:

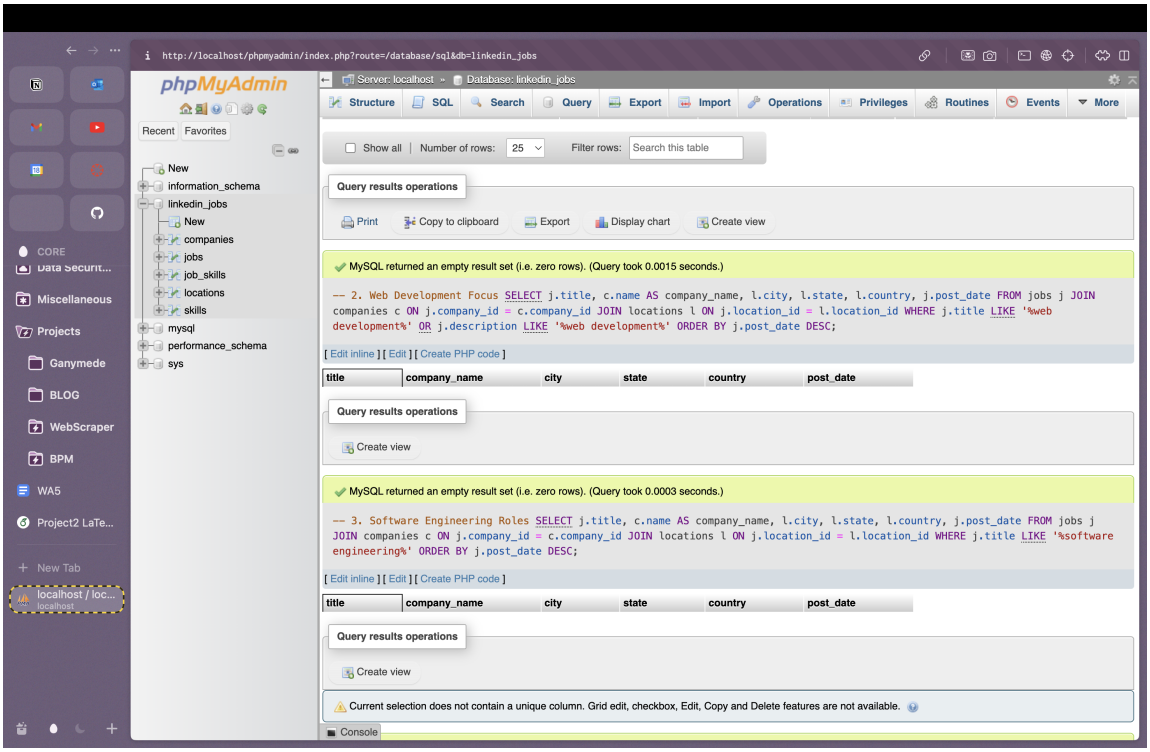

Figure 6: Query 3

3.4 Query 4: IT Position Search (Adjusted)

English:
Display all IT-related job postings, including job title, company name, location, and post date. **SQL:**

```
SELECT j.title, c.name AS company_name, l.city, l.state, l.country, j.post_date
FROM jobs j
JOIN companies c ON j.company_id = c.company_id
JOIN locations l ON j.location_id = l.location_id
WHERE j.title LIKE '%IT%' OR j.description LIKE '%IT%'
ORDER BY j.post_date DESC;
```

Screenshot:

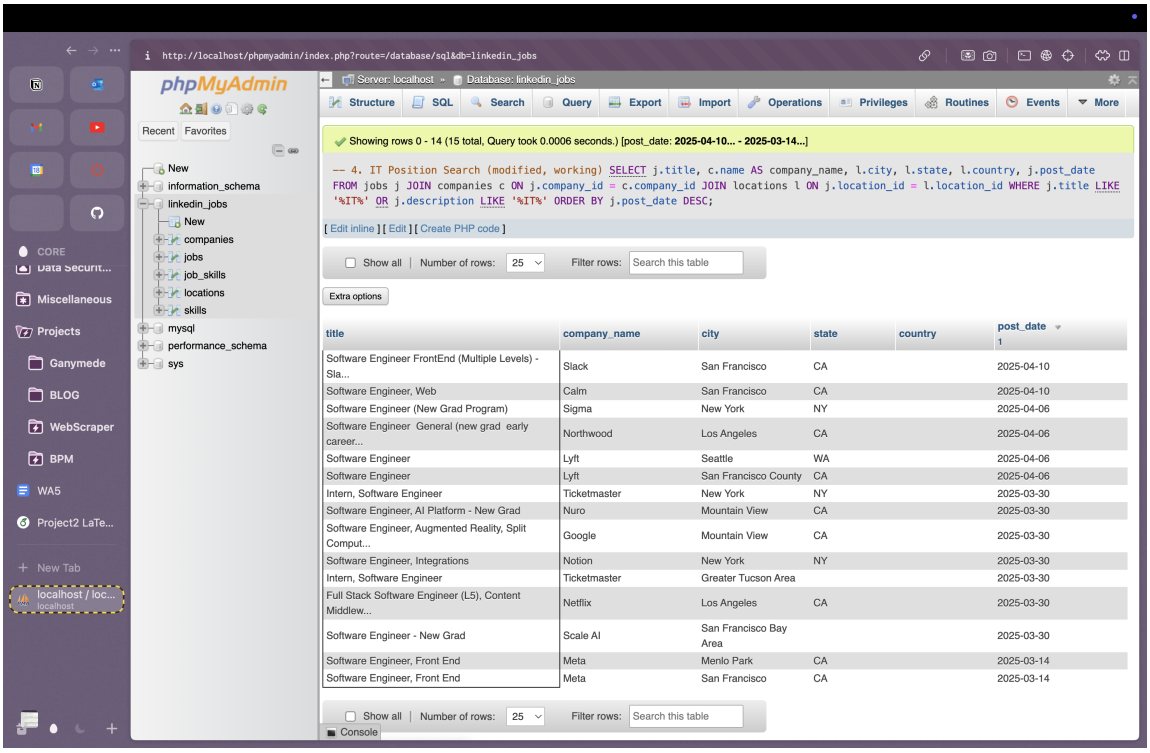

Figure 7: Query 4

3.5 Query 5: Front-End vs. Back-End Focus

English:

List job postings that include either 'front end' or 'back end' in the job title, with company name, location, and post date. **SQL:**

```
SELECT j.title, c.name AS company_name, l.city, l.state, l.country, j.post_date
FROM jobs j
JOIN companies c ON j.company_id = c.company_id
JOIN locations l ON j.location_id = l.location_id
WHERE j.title LIKE '%front end%' OR j.title LIKE '%back end%'
ORDER BY j.post_date DESC;
```

Screenshot:

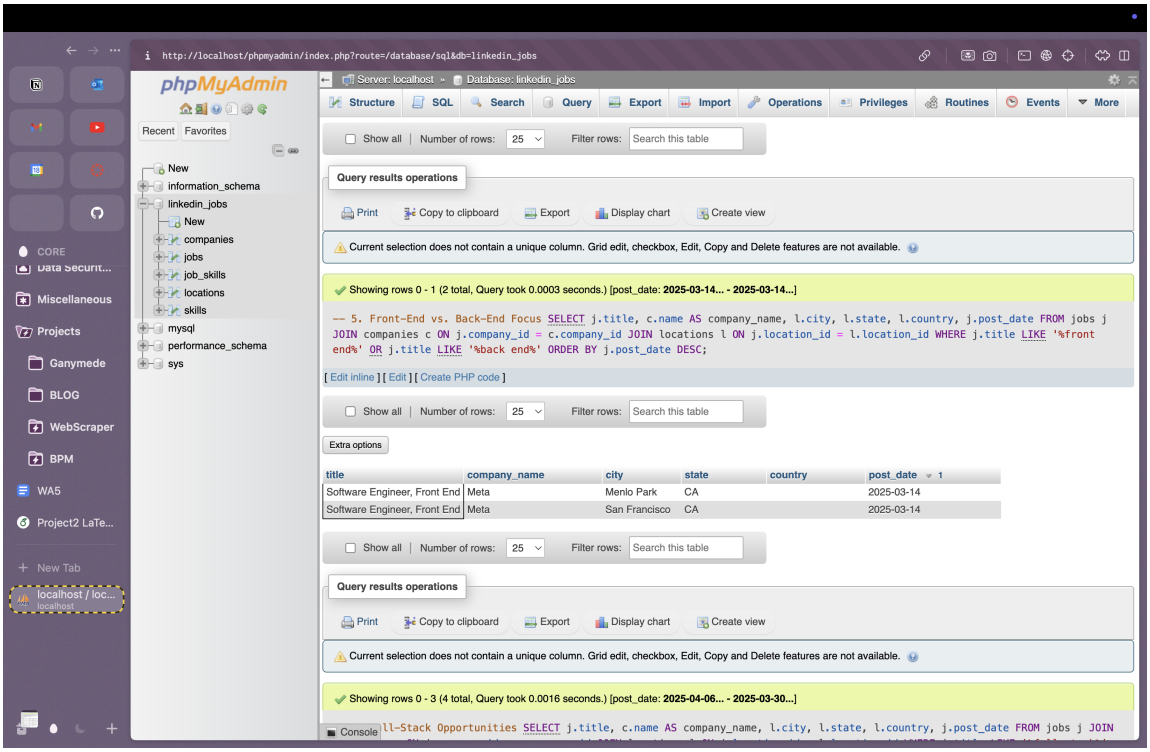

Figure 8: Query 5

3.6 Query 6: Full-Stack Opportunities

English:
Fetch job postings that mention 'full-stack' in the job title or description, displaying the job title, company name, and location. **SQL:**

```
SELECT j.title, c.name AS company_name, l.city, l.state, l.country
FROM jobs j
JOIN companies c ON j.company_id = c.company_id
JOIN locations l ON j.location_id = l.location_id
WHERE j.title LIKE '%full-stack%' OR j.description LIKE '%full-stack%'
ORDER BY j.post_date DESC;
```

Screenshot:

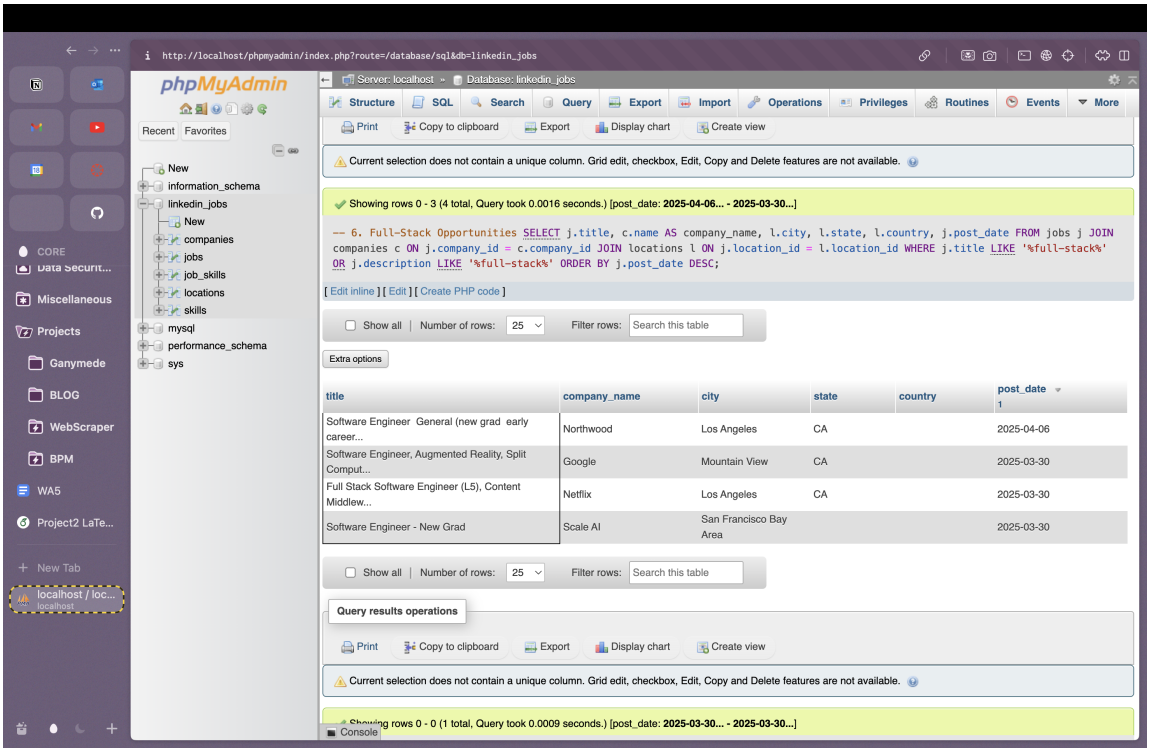

Figure 9: Query 6

3.7 Query 7: Tech Skills Filter

English:

Retrieve job postings where the associated skills include both 'React' and 'Node.js', showing the job title and company name. **SQL:**

```
SELECT DISTINCT j.title, c.name AS company_name
FROM jobs j
JOIN companies c ON j.company_id = c.company_id
JOIN job_skills js1 ON j.job_id = js1.job_id
JOIN skills s1 ON js1.skill_id = s1.skill_id
JOIN job_skills js2 ON j.job_id = js2.job_id
JOIN skills s2 ON js2.skill_id = s2.skill_id
WHERE s1.skill_name = 'React' AND s2.skill_name = 'Node.js'
ORDER BY j.post_date DESC;
```

Screenshot:

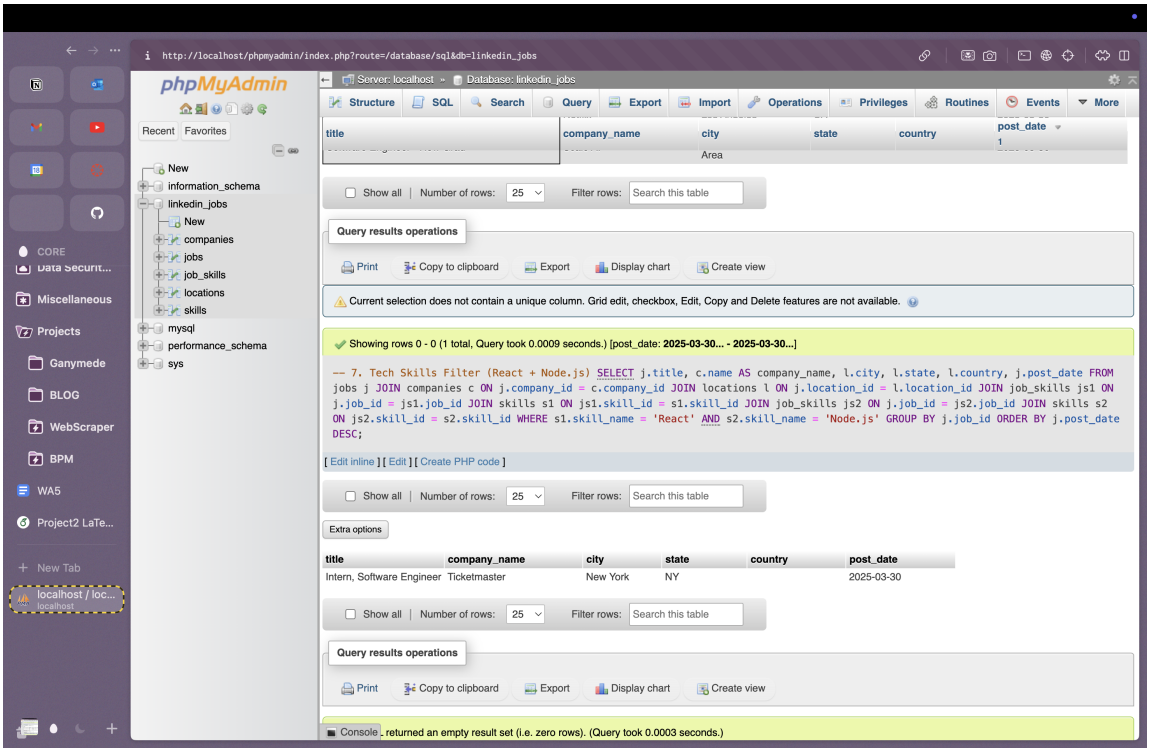

Figure 10: Query 7

3.8 Query 8: Seniority Filter

English:
List job postings for roles with seniority indicators like 'Senior' or 'Lead' in the title, along with company name, location, and post date. **SQL:**

```
SELECT j.title, c.name AS company_name, l.city, l.state, l.country, j.post_date
FROM jobs j
JOIN companies c ON j.company_id = c.company_id
JOIN locations l ON j.location_id = l.location_id
WHERE j.title LIKE '%Senior%' OR j.title LIKE '%Lead%'
ORDER BY j.post_date DESC;
```

Screenshot:

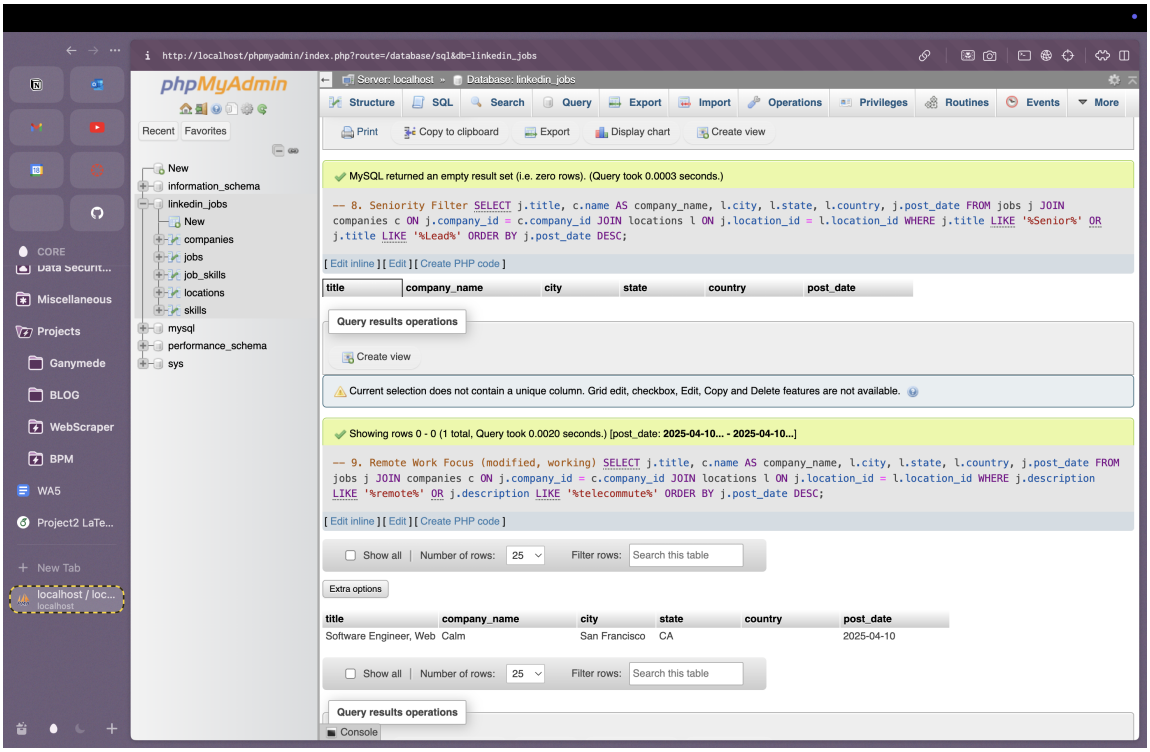

Figure 11: Query 8

3.9 Query 9: Remote Work Focus (Adjusted)

English:
Display all job postings that mention ’remote’ or ’telecommute’ in the description, including job title, company name, and post date. **SQL:**

```
SELECT j.title, c.name AS company_name, j.post_date
FROM jobs j
JOIN companies c ON j.company_id = c.company_id
WHERE j.description LIKE '%remote%' OR j.description LIKE '%telecommute%'
ORDER BY j.post_date DESC;
```

Screenshot:

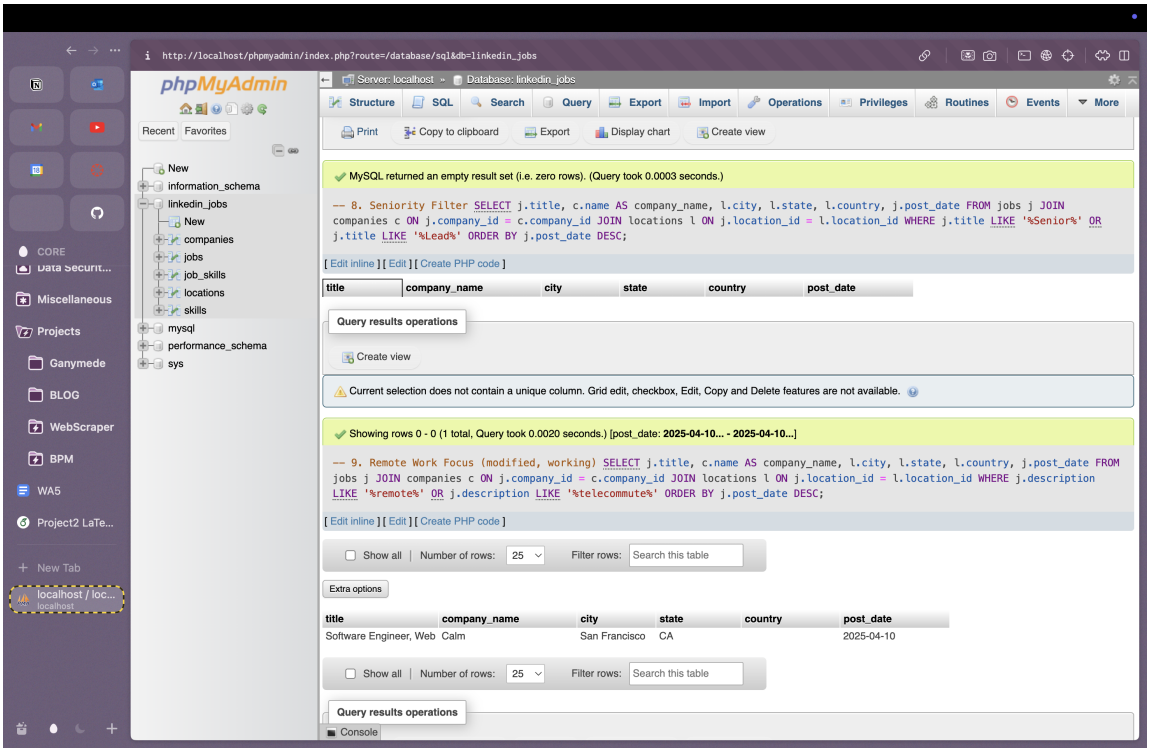

Figure 12: Query 9

3.10 Query 10: Category Summary Count

English:

Provide a summary count of job postings by category (e.g., web development, software engineering, IT, front-end, back-end, full-stack) based on keywords in the job title or description. **SQL:**

```
SELECT
CASE
    WHEN j.title LIKE '%web development%' OR j.description LIKE '%web development%' T
    WHEN j.title LIKE '%software engineering%' OR j.description LIKE '%software engin
    WHEN j.title LIKE '%IT%' OR j.description LIKE '%IT%' THEN 'IT'
    WHEN j.title LIKE '%front end%' OR j.description LIKE '%front end%' THEN 'Front E
    WHEN j.title LIKE '%back end%' OR j.description LIKE '%back end%' THEN 'Back End'
    WHEN j.title LIKE '%full-stack%' OR j.description LIKE '%full-stack%' THEN 'Full
    ELSE 'Other'
END AS category,
COUNT(*) AS job_count
```

```
FROM jobs j
GROUP BY category;
```

Screenshot:

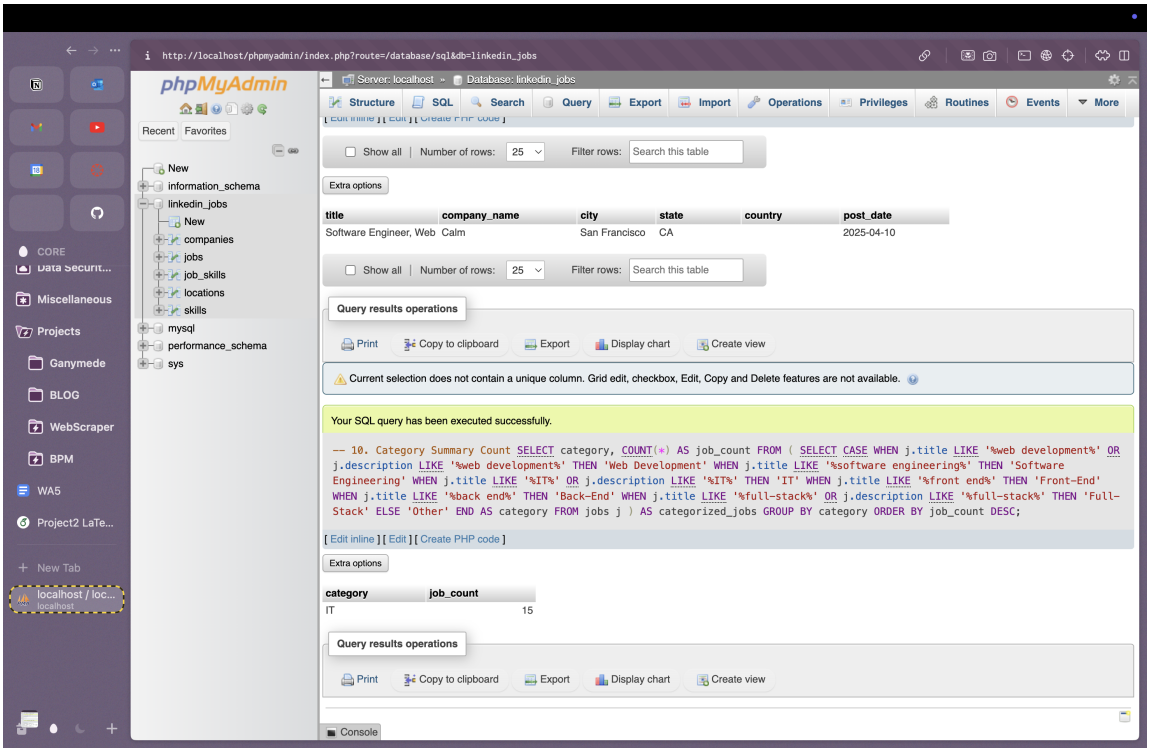

Figure 13: Query 10

4 Backend Accomplishments

- Clean and normalized database schema implemented
- Data stored dynamically in MySQL, visible in phpMyAdmin
- Python scraper successfully pulls live data from LinkedIn
- Parsing algorithm is hardcoded for precision
- CLI tool `query_db.py` works for interactive querying

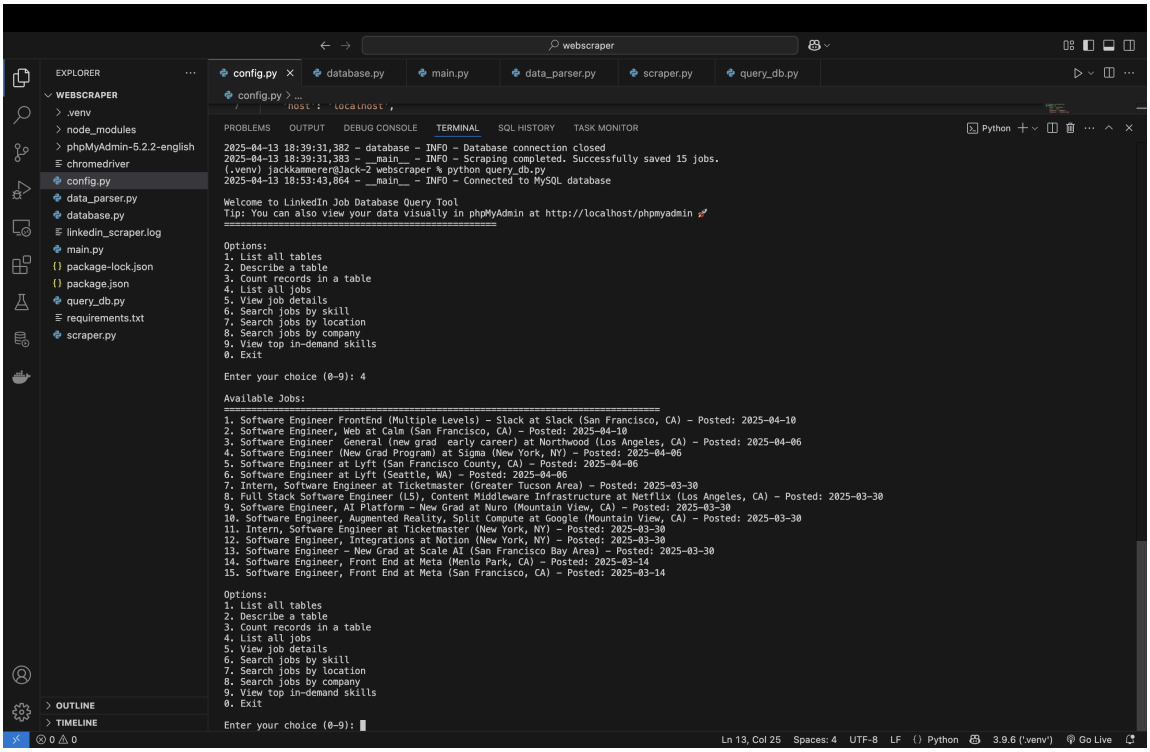

Figure 14: Developer File - To Test New Data

5 Source Code

5.1 Main Python Files

5.2 main.py

```
"""
Main execution script for LinkedIn job scraper.
This script orchestrates the entire scraping, parsing, and saving process.
"""

import sys
import logging
import time
from scraper import LinkedInScraper
from data_parser import process_job_data
from database import Database
from config import SAMPLE_URL

# Configure logging
```

```
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler("linkedin_scraper.log"),
        logging.StreamHandler(sys.stdout)
    ]
)
logger = logging.getLogger(__name__)


def run_scraper(url):
    """
    Run the complete LinkedIn job scraping process.

    Args:
        url (str): LinkedIn job search URL

    Returns:
        int: Number of jobs successfully scraped and saved
    """
    logger.info("Starting LinkedIn job scraper")

    db = None
    scraper = None
    job_count = 0

    try:
        # Initialize database
        db = Database()
        db.reset_database()
        logger.info("Database initialized and reset")

        # Initialize scraper
        scraper = LinkedInScraper(db, headless=True)
        logger.info("LinkedIn scraper initialized")

        # Scrape job listing URLs
        job_urls = scraper.scrape_job_listings(url)

        if not job_urls:
            logger.warning("No job listings found")
            return 0
```

```
58
59          logger.info(f"Found {len(job_urls)} job listings, proceeding to
    scrape details")
60
61          # Process each job URL
62          for job_url in job_urls:
63              try:
64                  # Scrape job details
65                  raw_job_data = scraper.scrape_job_details(job_url)
66
67                  if not raw_job_data or 'title' not in raw_job_data:
68                      logger.warning(f"Skipping job, could not scrape details
    : {job_url}")
69                      continue
70
71                  # Process and clean job data
72                  processed_job_data = process_job_data(raw_job_data)
73
74                  # Save to database
75                  db.save_job_listing(processed_job_data)
76                  job_count += 1
77
78                  logger.info(f"Successfully processed and saved job: {
    processed_job_data['title']}")
79
80                  # Random sleep between requests
81                  time.sleep(2)  # Small delay between jobs
82
83              except Exception as e:
84                  logger.error(f"Error processing job {job_url}: {e}")
85                  continue
86
87          logger.info(f"Completed job scraping. Successfully saved {job_count
    } jobs.")
88          return job_count
89
90      except Exception as e:
91          logger.error(f"Error in job scraping process: {e}")
92          return job_count
93
94      finally:
95          # Clean up resources
96          if scraper:
```

```
            scraper.close()

        if db:
            db.close()


if __name__ == "__main__":
    # Get URL from command line or use sample URL
    url = sys.argv[1] if len(sys.argv) > 1 else SAMPLE_URL

    logger.info(f"Starting job scraping with URL: {url}")
    success_count = run_scraper(url)

    logger.info(f"Scraping completed. Successfully saved {success_count}
    jobs.")
```

5.3 query_db.py

```
"""
Database query tool for LinkedIn job scraper.
This script helps query the normalized database tables.
"""

import mysql.connector
import logging
from config import DB_CONFIG

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

class QueryTool:
    def __init__(self):
        """Initialize database connection."""
        self.connection = None
        try:
            self.connection = mysql.connector.connect(**DB_CONFIG)
            if self.connection.is_connected():
                logger.info("Connected to MySQL database")
        except Exception as e:
            logger.error(f"Error connecting to MySQL: {e}")
```

```
            raise

    def list_tables(self):
        """List all tables in the database."""
        cursor = self.connection.cursor()
        try:
            cursor.execute("SHOW TABLES")
            tables = cursor.fetchall()
            print("\nAvailable tables:")
            for table in tables:
                print(f"- {table[0]}")
            return tables
        except Exception as e:
            logger.error(f"Error listing tables: {e}")
            return []
        finally:
            cursor.close()

    def describe_table(self, table_name):
        """Describe the structure of a specific table."""
        cursor = self.connection.cursor()
        try:
            cursor.execute(f"DESCRIBE {table_name}")
            columns = cursor.fetchall()
            print(f"\nTable structure for '{table_name}':")
            for col in columns:
                print(f"- {col[0]} ({col[1]})")
            return columns
        except Exception as e:
            logger.error(f"Error describing table: {e}")
            return []
        finally:
            cursor.close()

    def count_records(self, table_name):
        """Count the number of records in a table."""
        cursor = self.connection.cursor()
        try:
            cursor.execute(f"SELECT COUNT(*) FROM {table_name}")
            count = cursor.fetchone()[0]
            print(f"\nNumber of records in '{table_name}': {count}")
            return count
        except Exception as e:
```

```
            logger.error(f"Error counting records: {e}")
            return 0
        finally:
            cursor.close()

    def list_jobs(self):
        """List all jobs with company and location details."""
        cursor = self.connection.cursor(dictionary=True)
        try:
            query = """
            SELECT j.job_id, j.title, c.name AS company_name,
                   l.city, l.state, l.country, j.post_date
            FROM jobs j
            JOIN companies c ON j.company_id = c.company_id
            JOIN locations l ON j.location_id = l.location_id
            ORDER BY j.post_date DESC
            """
            cursor.execute(query)
            jobs = cursor.fetchall()

            print("\nAvailable Jobs:")
            print("=" * 80)
            for idx, job in enumerate(jobs, 1):
                location = ', '.join(filter(None, [job['city'], job['state'
], job['country']]))
                print(f"{idx}. {job['title']} at {job['company_name']} ({
location}) - Posted: {job['post_date']}")

            return jobs
        except Exception as e:
            logger.error(f"Error listing jobs: {e}")
            return []
        finally:
            cursor.close()

    def job_details(self, job_id):
        """Get detailed information about a specific job."""
        cursor = self.connection.cursor(dictionary=True)
        try:
            job_query = """
            SELECT j.job_id, j.title, c.name AS company_name, c.industry, c
.size AS company_size,
                   l.city, l.state, l.country, j.description, j.post_date
```

```
            FROM jobs j
            JOIN companies c ON j.company_id = c.company_id
            JOIN locations l ON j.location_id = l.location_id
            WHERE j.job_id = %s
            """
            cursor.execute(job_query, (job_id,))
            job = cursor.fetchone()

            if not job:
                print(f"No job found with ID {job_id}")
                return None

            # Get skills for this job
            skills_query = """
            SELECT s.skill_name
            FROM job_skills js
            JOIN skills s ON js.skill_id = s.skill_id
            WHERE js.job_id = %s
            """
            cursor.execute(skills_query, (job_id,))
            skills = [row['skill_name'] for row in cursor.fetchall()]

            # Print job details
            print("\nJob Details:")
            print("=" * 80)
            print(f"Title: {job['title']}")
            print(f"Company: {job['company_name']}")
            print(f"Industry: {job['industry'] or 'Not specified'}")
            print(f"Company Size: {job['company_size'] or 'Not specified'}"
)

            location = ', '.join(filter(None, [job['city'], job['state'],
job['country']]))
            print(f"Location: {location}")
            print(f"Posted on: {job['post_date']}")

            print("\nSkills:")
            if skills:
                for skill in skills:
                    print(f"- {skill}")
            else:
                print("No skills listed")

```

```
            print("\nDescription:")
            print("-" * 80)
            print(job['description'])

            return {**job, 'skills': skills}
        except Exception as e:
            logger.error(f"Error getting job details: {e}")
            return None
        finally:
            cursor.close()

    def search_jobs_by_skill(self, skill):
        """Search for jobs requiring a specific skill."""
        cursor = self.connection.cursor(dictionary=True)
        try:
            query = """
            SELECT j.job_id, j.title, c.name AS company_name,
                   l.city, l.state, l.country
            FROM jobs j
            JOIN companies c ON j.company_id = c.company_id
            JOIN locations l ON j.location_id = l.location_id
            JOIN job_skills js ON j.job_id = js.job_id
            JOIN skills s ON js.skill_id = s.skill_id
            WHERE s.skill_name LIKE %s
            ORDER BY j.post_date DESC
            """
            cursor.execute(query, (f"%{skill}%",))
            jobs = cursor.fetchall()

            print(f"\nJobs requiring '{skill}':")
            print("=" * 80)
            if jobs:
                for job in jobs:
                    location = ', '.join(filter(None, [job['city'], job['
state'], job['country']]))
                    print(f"{job['job_id']}. {job['title']} at {job['
company_name']} ({location})")
            else:
                print(f"No jobs found requiring '{skill}'")

            return jobs
        except Exception as e:
            logger.error(f"Error searching jobs by skill: {e}")
```

```
            return []
        finally:
            cursor.close()

    def search_jobs_by_location(self, location):
        """Search for jobs in a specific location."""
        cursor = self.connection.cursor(dictionary=True)
        try:
            query = """
            SELECT j.job_id, j.title, c.name AS company_name,
                   l.city, l.state, l.country
            FROM jobs j
            JOIN companies c ON j.company_id = c.company_id
            JOIN locations l ON j.location_id = l.location_id
            WHERE l.city LIKE %s OR l.state LIKE %s OR l.country LIKE %s
            ORDER BY j.post_date DESC
            """
            pattern = f"%{location}%"
            cursor.execute(query, (pattern, pattern, pattern))
            jobs = cursor.fetchall()

            print(f"\nJobs in '{location}':")
            print("=" * 80)
            if jobs:
                for job in jobs:
                    location = ', '.join(filter(None, [job['city'], job['
state'], job['country']]))
                    print(f"{job['job_id']}. {job['title']} at {job['
company_name']} ({location})")
            else:
                print(f"No jobs found in '{location}'")

            return jobs
        except Exception as e:
            logger.error(f"Error searching jobs by location: {e}")
            return []
        finally:
            cursor.close()

    def search_jobs_by_company(self, company):
        """Search for jobs at a specific company."""
        cursor = self.connection.cursor(dictionary=True)
        try:
```

```
            query = """
            SELECT j.job_id, j.title, c.name AS company_name,
                   l.city, l.state, l.country
            FROM jobs j
            JOIN companies c ON j.company_id = c.company_id
            JOIN locations l ON j.location_id = l.location_id
            WHERE c.name LIKE %s
            ORDER BY j.post_date DESC
            """
            cursor.execute(query, (f"%{company}%",))
            jobs = cursor.fetchall()

            print(f"\nJobs at '{company}':")
            print("=" * 80)
            if jobs:
                for job in jobs:
                    location = ', '.join(filter(None, [job['city'], job['state'], job['country']]))
                    print(f"{job['job_id']}. {job['title']} at {job['company_name']} ({location})")
            else:
                print(f"No jobs found at '{company}'")

            return jobs
        except Exception as e:
            logger.error(f"Error searching jobs by company: {e}")
            return []
        finally:
            cursor.close()

    def top_skills(self, limit=10):
        """List the most in-demand skills based on job listings."""
        cursor = self.connection.cursor(dictionary=True)
        try:
            query = """
            SELECT s.skill_name, COUNT(js.job_id) as job_count
            FROM skills s
            JOIN job_skills js ON s.skill_id = js.skill_id
            GROUP BY s.skill_name
            ORDER BY job_count DESC
            LIMIT %s
            """
            cursor.execute(query, (limit,))
```

```
            skills = cursor.fetchall()

            print(f"\nTop {limit} in-demand skills:")
            print("=" * 50)
            for i, skill in enumerate(skills, 1):
                print(f"{i}. {skill['skill_name']} - Found in {skill['job_count']} job(s)")

            return skills
        except Exception as e:
            logger.error(f"Error getting top skills: {e}")
            return []
        finally:
            cursor.close()

    def close(self):
        """Close the database connection."""
        if self.connection and self.connection.is_connected():
            self.connection.close()
            logger.info("Database connection closed")


if __name__ == "__main__":
    tool = QueryTool()

    try:
        print("\nWelcome to LinkedIn Job Database Query Tool")
        print("Tip: You can also view your data visually in phpMyAdmin at http://localhost/phpmyadmin     ")
        print("=" * 50)

        while True:
            print("\nOptions:")
            print("1. List all tables")
            print("2. Describe a table")
            print("3. Count records in a table")
            print("4. List all jobs")
            print("5. View job details")
            print("6. Search jobs by skill")
            print("7. Search jobs by location")
            print("8. Search jobs by company")
            print("9. View top in-demand skills")
            print("0. Exit")
```

```
315
316             choice = input("\nEnter your choice (0-9): ")
317
318             if choice == '0':
319                 print("Exiting...")
320                 break
321             elif choice == '1':
322                 tool.list_tables()
323             elif choice == '2':
324                 table = input("Enter table name: ")
325                 tool.describe_table(table)
326             elif choice == '3':
327                 table = input("Enter table name: ")
328                 tool.count_records(table)
329             elif choice == '4':
330                 tool.list_jobs()
331             elif choice == '5':
332                 job_id = input("Enter job ID: ")
333                 tool.job_details(int(job_id))
334             elif choice == '6':
335                 skill = input("Enter skill to search: ")
336                 tool.search_jobs_by_skill(skill)
337             elif choice == '7':
338                 location = input("Enter location to search: ")
339                 tool.search_jobs_by_location(location)
340             elif choice == '8':
341                 company = input("Enter company to search: ")
342                 tool.search_jobs_by_company(company)
343             elif choice == '9':
344                 try:
345                     limit = int(input("Enter number of skills to show (
    default 10): ") or 10)
346                     tool.top_skills(limit)
347                 except ValueError:
348                     print("Invalid input. Using default limit of 10.")
349                     tool.top_skills()
350             else:
351                 print("Invalid choice. Please try again.")
352
353     except Exception as e:
354         logger.error(f"An error occurred: {e}")
355
356     finally:
```

```
357         tool.close()
```

5.4 scraper.py

```
"""
LinkedIn job scraper module.
This module handles the web scraping functionality.
"""

import time
import random
import logging
import mysql.connector
from config import DB_CONFIG
from database import Database
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.common.exceptions import (
    TimeoutException, NoSuchElementException,
    StaleElementReferenceException
)

from config import SCRAPING_CONFIG, USER_AGENT

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)


class LinkedInScraper:
    def __init__(self, db, headless=True):
        """
        Initialize the LinkedIn scraper.

        Args:
            headless (bool): Whether to run the browser in headless mode
        """
```

```
        self.db = db
        self.setup_driver(headless)

    def setup_driver(self, headless):
        """
        Set up the Selenium WebDriver.

        Args:
            headless (bool): Whether to run in headless mode
        """
        chrome_options = Options()
        if headless:
            chrome_options.add_argument("--headless")

        chrome_options.add_argument("--no-sandbox")
        chrome_options.add_argument("--disable-dev-shm-usage")
        chrome_options.add_argument("--disable-gpu")
        chrome_options.add_argument(f"user-agent={USER_AGENT}")

        service = Service('./chromedriver')
        self.driver = webdriver.Chrome(service=service, options=
    chrome_options)

        try:
            self.driver.set_window_size(1920, 1080)
            self.wait = WebDriverWait(self.driver, SCRAPING_CONFIG['timeout
    '])
            logger.info("WebDriver initialized successfully")
        except Exception as e:
            logger.error(f"Failed to initialize WebDriver: {e}")
            raise

    def scrape_job_listings(self, url):
        """
        Scrape job listings from a LinkedIn search URL.

        Args:
            url (str): LinkedIn job search URL

        Returns:
            list: List of job listing URLs
        """
        logger.info(f"Starting to scrape job listings from: {url}")
```

```
        job_urls = []

        try:
            self.driver.get(url)
            time.sleep(random.uniform(3, 5))  # Initial wait for page to load

            # Get job cards
            job_cards = self.wait.until(
                EC.presence_of_all_elements_located(
                    (By.CSS_SELECTOR, ".jobs-search__results-list li")
                )
            )

            # Extract job URLs
            count = 0
            for card in job_cards:
                if count >= SCRAPING_CONFIG['max_jobs']:
                    break

                try:
                    job_link = card.find_element(By.CSS_SELECTOR, "a").get_attribute("href")
                    job_urls.append(job_link)
                    count += 1
                except (NoSuchElementException, StaleElementReferenceException) as e:
                    logger.warning(f"Could not extract job link: {e}")
                    continue

            logger.info(f"Found {len(job_urls)} job listings")
            return job_urls

        except TimeoutException:
            logger.error("Timeout while loading LinkedIn job search results")
            return []
        except Exception as e:
            logger.error(f"Error scraping job listings: {e}")
            return []

    def scrape_job_details(self, job_url):
        """
```

```
        Scrape detailed information for a specific job.

        Args:
            job_url (str): URL of the job listing

        Returns:
            dict: Job details including title, company, location, etc.
        """
        logger.info(f"Scraping details for job: {job_url}")
        job_data = {}

        try:
            self.driver.get(job_url)
            time.sleep(random.uniform(*SCRAPING_CONFIG['sleep_interval']))

            job_data['title'] = self.safe_extract(By.CSS_SELECTOR, "h1.
    topcard__title")
            job_data['company_name'] = self.safe_extract(By.CSS_SELECTOR, "
    a.topcard__org-name-link")
            location = self.safe_extract(By.CSS_SELECTOR, "span.
    topcard__flavor.topcard__flavor--bullet")
            if location:
                location_parts = location.split(',')
                job_data['city'] = location_parts[0].strip() if len(
    location_parts) > 0 else None
                job_data['state'] = location_parts[1].strip() if len(
    location_parts) > 1 else None
                job_data['country'] = location_parts[-1].strip() if len(
    location_parts) > 2 else None

            job_data['description'] = self.safe_extract(By.CSS_SELECTOR, "
    div.description__text", get_text=True)
            job_data['post_date'] = self.safe_extract(By.CSS_SELECTOR, "
    span.posted-time-ago__text")

            skills_elements = self.driver.find_elements(By.CSS_SELECTOR, ".
    skill-tag")
            job_data['skills'] = [skill.text for skill in skills_elements
    if skill.text]

            job_data['company_size'] = self.safe_extract(By.CSS_SELECTOR, "
    .company-size")
```

```
            job_data['industry'] = self.safe_extract(By.CSS_SELECTOR, ".company-industry")

            logger.info(f"Successfully scraped job details for: {job_data.get('title', 'Unknown job')}")
            return job_data

        except TimeoutException:
            logger.error(f"Timeout while loading job details: {job_url}")
            return {}
        except Exception as e:
            logger.error(f"Error scraping job details: {e}")
            return {}

    def safe_extract(self, by, selector, get_text=False):
        """
        Safely extract an element's text or attribute.

        Args:
            by: Selenium By locator
            selector (str): CSS selector
            get_text (bool): Whether to get innerText instead of textContent

        Returns:
            str: Extracted text or None if element not found
        """
        try:
            element = self.driver.find_element(by, selector)
            if get_text:
                return element.get_attribute("innerText")
            return element.text
        except (NoSuchElementException, StaleElementReferenceException):
            return None

    def close(self):
        """Close the WebDriver."""
        if self.driver:
            self.driver.quit()
            logger.info("WebDriver closed")
```

5.5 database.py

```
"""
```

```
Database operations for the LinkedIn job scraper.
This module handles MySQL connections and database queries.
"""

import mysql.connector
from mysql.connector import Error
import logging
from config import DB_CONFIG

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)


class Database:
    def __init__(self):
        """Initialize database connection and create tables if they don't exist."""
        self.connection = None
        try:
            self.connection = mysql.connector.connect(**DB_CONFIG)
            if self.connection.is_connected():
                logger.info("Connected to MySQL database")
                self.create_tables()
        except Error as e:
            logger.error(f"Error connecting to MySQL: {e}")
            raise

    def reset_database(self):
        """
        Drop all tables and recreate them to start fresh.
        """
        cursor = self.connection.cursor()
        try:
            logger.info("Resetting database...")

            # Disable foreign key checks to avoid constraint errors during drop
            cursor.execute("SET FOREIGN_KEY_CHECKS = 0;")

```

```
            # Drop tables if they exist
            tables = ['job_skills', 'jobs', 'skills', 'locations', '
companies']
            for table in tables:
                cursor.execute(f"DROP TABLE IF EXISTS {table};")
                logger.info(f"Dropped table {table}")

            # Re-enable foreign key checks
            cursor.execute("SET FOREIGN_KEY_CHECKS = 1;")

            # Recreate the tables
            self.create_tables()
            logger.info("Database reset and tables recreated successfully."
)

        except Error as e:
            logger.error(f"Error resetting database: {e}")
            raise
        finally:
            cursor.close()

    def create_tables(self):
        """Create necessary tables if they don't exist."""
        cursor = self.connection.cursor()
        try:
            # Create companies table
            cursor.execute('''
                CREATE TABLE IF NOT EXISTS companies (
                    company_id INT AUTO_INCREMENT PRIMARY KEY,
                    name VARCHAR(255) NOT NULL,
                    industry VARCHAR(255),
                    size VARCHAR(100),
                    UNIQUE (name)
                )
            ''')

            # Create locations table
            cursor.execute('''
                CREATE TABLE IF NOT EXISTS locations (
                    location_id INT AUTO_INCREMENT PRIMARY KEY,
                    city VARCHAR(100),
                    state VARCHAR(100),
                    country VARCHAR(100),
```

```
                    UNIQUE (city, state, country)
                )
            ''')

            # Create skills table
            cursor.execute('''
                CREATE TABLE IF NOT EXISTS skills (
                    skill_id INT AUTO_INCREMENT PRIMARY KEY,
                    skill_name VARCHAR(100) NOT NULL,
                    UNIQUE (skill_name)
                )
            ''')

            # Create jobs table
            cursor.execute('''
                CREATE TABLE IF NOT EXISTS jobs (
                    job_id INT AUTO_INCREMENT PRIMARY KEY,
                    title VARCHAR(255) NOT NULL,
                    company_id INT,
                    location_id INT,
                    description TEXT,
                    post_date DATE,
                    FOREIGN KEY (company_id) REFERENCES companies (company_id),
                    FOREIGN KEY (location_id) REFERENCES locations (location_id)
                )
            ''')

            # Create job_skills join table
            cursor.execute('''
                CREATE TABLE IF NOT EXISTS job_skills (
                    job_id INT,
                    skill_id INT,
                    PRIMARY KEY (job_id, skill_id),
                    FOREIGN KEY (job_id) REFERENCES jobs (job_id) ON DELETE CASCADE,
                    FOREIGN KEY (skill_id) REFERENCES skills (skill_id) ON DELETE CASCADE
                )
            ''')

            self.connection.commit()
```

```
            logger.info("Database tables created successfully")
        except Error as e:
            logger.error(f"Error creating tables: {e}")
            raise
        finally:
            cursor.close()

    def insert_company(self, name, industry=None, size=None):
        """
        Insert a company or get its ID if it already exists.

        Args:
            name (str): Company name
            industry (str, optional): Company industry
            size (str, optional): Company size

        Returns:
            int: The company_id
        """
        cursor = self.connection.cursor()
        try:
            # Check if company already exists
            cursor.execute("SELECT company_id FROM companies WHERE name = %
s", (name,))
            result = cursor.fetchone()

            if result:
                return result[0]

            # Insert new company
            cursor.execute(
                "INSERT INTO companies (name, industry, size) VALUES (%s, %
s, %s)",
                (name, industry, size)
            )
            self.connection.commit()
            return cursor.lastrowid
        except Error as e:
            self.connection.rollback()
            logger.error(f"Error inserting company: {e}")
            raise
        finally:
            cursor.close()
```

```
164
165     def insert_location(self, city=None, state=None, country=None):
166         """
167         Insert a location or get its ID if it already exists.
168
169         Args:
170             city (str, optional): City name
171             state (str, optional): State name
172             country (str, optional): Country name
173
174         Returns:
175             int: The location_id
176         """
177         cursor = self.connection.cursor()
178         try:
179             # Check if location already exists
180             cursor.execute(
181                 "SELECT location_id FROM locations WHERE city = %s AND
    state = %s AND country = %s",
182                 (city, state, country)
183             )
184             result = cursor.fetchone()
185
186             if result:
187                 return result[0]
188
189             # Insert new location
190             cursor.execute(
191                 "INSERT INTO locations (city, state, country) VALUES (%s, %
    s, %s)",
192                 (city, state, country)
193             )
194             self.connection.commit()
195             return cursor.lastrowid
196         except Error as e:
197             self.connection.rollback()
198             logger.error(f"Error inserting location: {e}")
199             raise
200         finally:
201             cursor.close()
202
203     def insert_skill(self, skill_name):
204         """
```

```
205         Insert a skill or get its ID if it already exists.
206 
207         Args:
208             skill_name (str): Name of the skill
209 
210         Returns:
211             int: The skill_id
212         """
213         cursor = self.connection.cursor()
214         try:
215             # Check if skill already exists
216             cursor.execute("SELECT skill_id FROM skills WHERE skill_name =
    %s", (skill_name,))
217             result = cursor.fetchone()
218 
219             if result:
220                 return result[0]
221 
222             # Insert new skill
223             cursor.execute("INSERT INTO skills (skill_name) VALUES (%s)", (
    skill_name,))
224             self.connection.commit()
225             return cursor.lastrowid
226         except Error as e:
227             self.connection.rollback()
228             logger.error(f"Error inserting skill: {e}")
229             raise
230         finally:
231             cursor.close()
232 
233     def insert_job(self, title, company_id, location_id, description,
    post_date):
234         """
235         Insert a job listing.
236 
237         Args:
238             title (str): Job title
239             company_id (int): Foreign key to companies table
240             location_id (int): Foreign key to locations table
241             description (str): Job description
242             post_date (str): Date job was posted
243 
244         Returns:
```

```
            int: The job_id
        """
        cursor = self.connection.cursor()
        try:
            cursor.execute(
                "INSERT INTO jobs (title, company_id, location_id,
    description, post_date) "
                "VALUES (%s, %s, %s, %s, %s)",
                (title, company_id, location_id, description, post_date)
            )
            self.connection.commit()
            return cursor.lastrowid
        except Error as e:
            self.connection.rollback()
            logger.error(f"Error inserting job: {e}")
            raise
        finally:
            cursor.close()

    def link_job_skill(self, job_id, skill_id):
        """
        Create a job-skill relationship in the join table.

        Args:
            job_id (int): ID of the job
            skill_id (int): ID of the skill
        """
        cursor = self.connection.cursor()
        try:
            cursor.execute(
                "INSERT INTO job_skills (job_id, skill_id) VALUES (%s, %s)"
    ,
                (job_id, skill_id)
            )
            self.connection.commit()
        except Error as e:
            self.connection.rollback()
            logger.error(f"Error linking job and skill: {e}")
            raise
        finally:
            cursor.close()

    def save_job_listing(self, job_data):
```

```
        """
        Save a complete job listing to the database.

        Args:
            job_data (dict): Dictionary containing job information

        Returns:
            int: The job_id of the inserted job
        """
        try:
            # Insert company and get ID
            company_id = self.insert_company(
                job_data['company_name'],
                job_data.get('industry'),
                job_data.get('company_size')
            )

            # Insert location and get ID
            location_id = self.insert_location(
                job_data.get('city'),
                job_data.get('state'),
                job_data.get('country')
            )

            # Insert job and get ID
            job_id = self.insert_job(
                job_data['title'],
                company_id,
                location_id,
                job_data.get('description', ''),
                job_data.get('post_date')
            )

            # Insert skills and link to job
            for skill in job_data.get('skills', []):
                skill_id = self.insert_skill(skill)
                self.link_job_skill(job_id, skill_id)

            logger.info(f"Successfully saved job: {job_data['title']} at {
    job_data['company_name']}")
            return job_id

        except Error as e:
```

```
            logger.error(f"Error saving job listing: {e}")
            raise

    def close(self):
        """Close the database connection."""
        if self.connection and self.connection.is_connected():
            self.connection.close()
            logger.info("Database connection closed")
```

5.6 config.py

```
"""
Configuration settings for LinkedIn job scraper.
"""

# Database configuration
DB_CONFIG = {
    'host': 'localhost',
    'user': 'root',
    'password': 'swagdata',
    'database': 'linkedin_jobs'
}

# Scraping configuration
SCRAPING_CONFIG = {
    'timeout': 30,  # Maximum seconds to wait for page loads
    'jobs_per_page': 25,
    'max_jobs': 15,  # Maximum number of jobs to scrape
    'sleep_interval': (2, 5),  # Random sleep interval between requests (min, max)
}

# User agent for requests
USER_AGENT = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36"

# Sample LinkedIn job search URL
SAMPLE_URL = "https://www.linkedin.com/jobs/search/?keywords=software%20engineer&location=United%20States"
```

6 Future Implementations

As I move forward with this project, my focus will be on refining and finalizing key components to ensure full functionality and presentation readiness. Specifically, I will:

- Finalize and test all SQL queries directly within phpMyAdmin to ensure accurate results
- Export the populated database schema for backup and grading purposes
- Prepare final screenshots of database tables, E/R diagram, and SQL query outputs
- Integrate English queries clearly mapped to their SQL counterparts within the report
- Insert all source code files into the final LaTeX report for complete project documentation
- Complete the final report and assemble the presentation deck

Additional Improvements:

- Fine-tune the parsing and scraping algorithms for better data consistency and accuracy
- Focus on UI/UX improvements to better showcase the scraped data and query results
- Consider adding pagination to enable scraping larger sets of job postings
- Optionally develop a Flask web interface for interactive data presentation
- Explore data export functionality for personal use in job applications

7 Conclusion

This milestone marks the most significant technical breakthrough in my project. After countless hours of trial and error, system restructures, and deep debugging, I successfully stabilized the backend functionality. Throughout the process, I experimented with multiple AI models, including smaller local LLMs and cloud-based tools, to assist with parsing and data manipulation. Despite initial setbacks, I realized that fully hardcoding my parsing algorithms and carefully mapping the data flow was the most reliable solution. I also navigated complex software dependencies, shifting between environments, and fine-tuned my schema design to meet full third normal form (3NF) standards. The database is now fully normalized and integrated with my DBMS (phpMyAdmin), providing clean, maintainable

data storage. This work required rethinking my initial design multiple times, significantly expanding from a simple two-entity system to a robust five-entity structure. With the core system now operational, I am positioned to finalize my SQL queries, capture reporting screenshots, and complete the final documentation for submission.

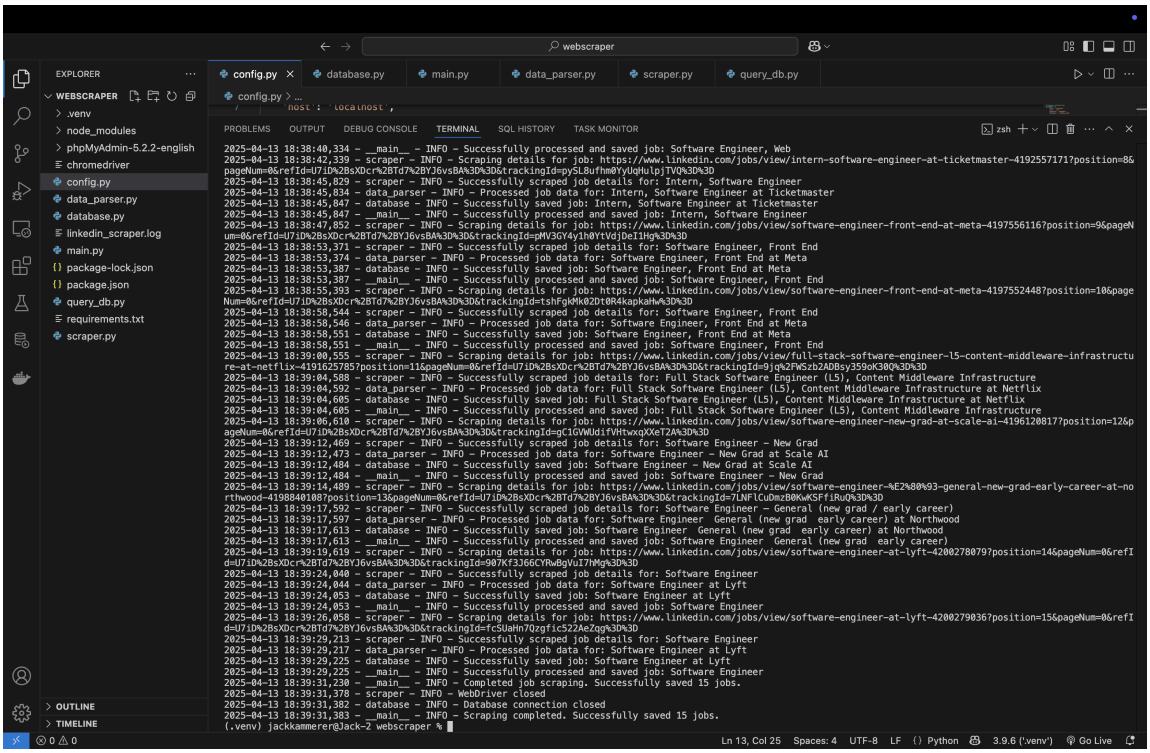

Figure 15: Finished Execution