Milestone 1 Report

Vincent Chen, Eunseo Lee, BK Kang, Andrew Lu

Our application is a database-driven web application tailored towards students (Waterloo co-op students are the users of the application) at the University of Waterloo to allow them to explore co-op salaries across a variety of companies, locations, roles/positions, terms, etc, which helps students make informed decisions on which companies and positions they would apply and compare salaries across programs and over time. One of the datasets from the internet we will use is a self-reported co-op salaries datasheet retrieved from a Reddit thread. Additionally we will use similar internship datasets from Kaggle and additional websites to obtain a large enough production dataset to demonstrate the improvement in performance for certain queries. The dataset from Reddit contains companies that offer at least one job posting on WaterlooWorks along with their hourly salary and work term number, and additional information such as benefits, year of job posting, and any companies put on a separate blacklist. However since the hourly salary for a company is likely to be an aggregation of individual job positions/roles within the company and many records within the dataset are empty, we will transform the Reddit dataset into a compatible relational format and populate the empty entries with synthetically generated numbers that reflects the average values observed in the current dataset. This populated and transformed dataset will become our production dataset. We (the project group) will be the administrators of the database system and will maintain the performance and availability of the database, ensure backup and recovery, handle blacklisted companies (via constraints and triggers), etc. The functionalities we would like to support are: listing salaries by program, faculty, or location, list top-paying companies by role/position, display salary across terms, flag companies with very low salary, provide table and chart-based outputs etc.

The system support for our application primarily uses Python with the Flask framework for the backend because it is a lightweight framework used for building web applications and is easily integrated with databases. Our application can be run on cross-platform OS such as MacOS, Windows, Linux, or Ubuntu, because these OSes support Flask natively and will facilitate each group member's development on parts of the application. We will use MySQL as our DBMS because MySQL supports the relational data schema we will use, is fast in query processing, allows for advanced features such as indexes, triggers, procedures, etc, and is compatible with the Flask framework through the *mysql-connector* library in Python. Additionally, for data processing we will use the *pandas* library for handling CSV files and formatting the data into relational structure and use *matplotlib* for visualizing the data into a graphical output. For deployment and testing, we will do so on the school server since it facilitates the hosting of our project without too much overhead. Finally, we will use Javascript, HTML, CSS, and React as the lightweight interface for the frontend of our application; it will have interactive elements such as dropdown menus, filters, buttons, etc.

We will get the sample data used to populate our database by sourcing them from publicly available datasets that have information related to internship and co-op opportunities. Our primary sources for our datasets are the self-reported co-op salaries datasheet from WaterlooWorks co-op postings and Kaggle datasets on internships. We will select a small subset of records from these datasets (around 100-200) for initial testing and format them as appropriate for testing our database in the early stages. Later on, our production dataset will incorporate more public datasets from the web and add in synthetically generated data to fill in certain entries. To populate our database with the sample data, we used our own synthetic data because it is easier for testing our database in the early stages and only requires inserting records via INSERT INTO statements in the .sql file; we may also use a Python script to insert more synthetically generated records into our database for further testing downstream.

The four members of our team and the work they did are:

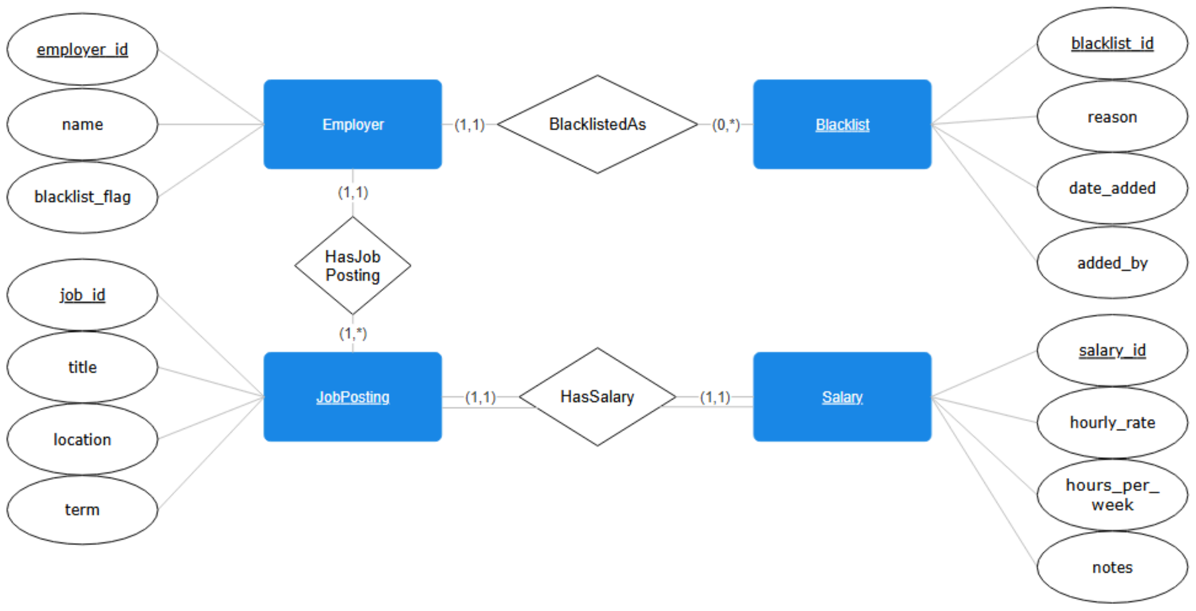
- Vincent Chen: Worked on designing the database schema, E/R diagram, all of R5, and wrote SQL files for creating tables, constraints, and triggers, etc (C2), and R6.
- Eunseo Lee: Worked on test-sample.sql file for Task 5 and the test-sample.out for testing our sample database (C3) and R8.
- BK Kang: Worked on setting up the MySQL and Flask environment for our project on GitHub, built the beginning parts of both the frontend and backend of our application, and implemented simple functionalities regarding the database (C1, C5) and R9.
- Andrew Lu: Worked on the Milestone 1 Report (R1, R2, R3) and R7.

GitHub repository: https://github.com/bkctrl/cs348-project

R5a. Assumptions about the data being modeled

1. **Employer**
   - Each employer has a unique name.
   - An employer may have multiple job postings.
   - Employers can be flagged in a separate `Blacklist` table for various reasons (e.g., poor working conditions, pay issues).
   - The `blacklist_flag` in `Employer` serves as a quick indicator, while `Blacklist` stores detailed records.
2. **JobPosting**
   - Each job posting belongs to exactly one employer.
   - Job titles and locations are descriptive and not normalized (for readability and simplicity).
   - A term (e.g., “Winter 2025”) identifies when the position was offered.
   - The same employer may post multiple jobs across different terms or locations.
3. **Salary**
   - Each salary record is tied to one job posting.
   - Salaries are expressed as hourly rates and must be non-negative.
   - Weekly hours are constrained between 0 and 80.
   - The `notes` field allows flexibility for extra data (e.g., comments, bonuses, or range notes).
4. **Blacklist**
   - Multiple blacklist records can exist for a single employer (different reasons or dates).
   - Each blacklist entry has a reason, the date added, and optionally who added it.
   - When an employer appears in `Blacklist`, its `blacklist_flag` in `Employer` should be set to TRUE.
   - Blacklist management is handled through controlled admin actions (not user-facing).
5. **General**
   - All primary keys are surrogate `INT AUTO_INCREMENT`.
   - Deleting an employer will cascade to its job postings and salary data (if enabled).
   - This system models **employers** → **jobs** → **salaries**, with **blacklist** as a supporting relationship.
   - No personally identifiable student information is stored.

R5b. E/R Model



R5b. Relational Data Model

Employer

Attribute	Type	Key	Description
employer_id	INT	**Primary Key**	Unique identifier for each employer
name	VARCHAR(255)	UNIQUE	Employer name (unique)
blacklist_flag	BOOLEAN		TRUE if the employer has any blacklist records

Notes:

- Each employer may have zero or more job postings.
- Each employer may have multiple blacklist entries.

JobPosting

Attribute	Type	Key	Description
job_id	INT	**Primary Key**	Unique ID for each job posting
employer_id	INT	**Foreign Key → Employer(employer_id)**	Employer who created the posting

title	VARCHAR(255)		Job title
location	VARCHAR(100)		City or region of the posting
term	VARCHAR(20)		Academic term (e.g., “Winter 2025”)

Notes:

- Each job posting belongs to exactly one employer.
- Each job posting has exactly one salary entry.

Salary

Attribute	Type	Key	Description
salary_id	INT	**Primary Key**	Unique salary record ID
job_id	INT	**Foreign Key → JobPosting(job_id)**	Related job posting
hourly_rate	DECIMAL(6,2)		Hourly pay rate (≥ 0)
hours_per_week	INT		Typical weekly hours (0–80)
notes	TEXT		Additional salary notes (e.g., bonuses, comments)

Notes:

- Every salary is tied to one job posting.
- Enforces valid numeric ranges for rate and hours.

Blacklist

Attribute	Type	Key	Description
blacklist_id	INT	**Primary Key**	Unique blacklist record
employer_id	INT	**Foreign Key → Employer(employer_id)**	Employer being blacklisted
reason	TEXT		Explanation for the blacklist entry
date_added	DATE		Defaults to current date
added_by	VARCHAR(100)		Admin or user who submitted the record

Notes:

- One employer may have multiple blacklist entries.
- Blacklist entries can exist independently of the `blacklist_flag` boolean in `Employer`, but should be synchronized via triggers.

Relationship Summary

Relationship	Participating Entities	Cardinality	Implementation
HasJobPosting	Employer – JobPosting	(1,1) → (1,n)	`JobPosting.employer_id` FK
HasSalary	JobPosting – Salary	(1,1) → (1,1)	`Salary.job_id` FK
BlacklistedAs	Employer – Blacklist	(1,1) → (0,n)	`Blacklist.employer_id` FK

R6a. Feature Interface Design
Feature Name: Keyword-Based Job Search
User: General user (student or visitor)

Description:
Users can search for co-op jobs by typing keywords into a search bar. The keywords are matched against the job title, employer name, and location fields.

User Interaction Flow:
1. The user navigates to the “Search Jobs” page.
2. A text box labeled “Search by keyword” is displayed.
3. The user enters a keyword such as “software”, “analyst”, or “Toronto”.
4. Upon clicking the Search button, the Flask app executes a parameterized SQL query against the database.
5. The results are displayed in a table showing: Job Title, Employer, Location, Term, and Hourly Rate.

INTERNAL
```
@app.route("/search", methods=["GET"])
def search():
    keyword = request.args.get("q", "")
    conn = get_db_connection()
    cursor = conn.cursor(dictionary=True)
    cursor.execute("""
        SELECT j.title, e.name AS employer, j.location, j.term, s.hourly_rate
        FROM JobPosting j
        JOIN Employer e ON j.employer_id = e.employer_id
        JOIN Salary s ON j.job_id = s.job_id
        WHERE j.title LIKE %s OR e.name LIKE %s OR j.location LIKE %s;
    """, (f"%{keyword}%", f"%{keyword}%", f"%{keyword}%"))
    rows = cursor.fetchall()
    conn.close()
    return render_template("search_results.html", jobs=rows)
```

R6b. SQL Query, Testing with Sample Data
SQL Query:
```
SELECT j.title, e.name AS employer, j.location, j.term, s.hourly_rate
FROM JobPosting j
JOIN Employer e ON j.employer_id = e.employer_id
JOIN Salary s ON j.job_id = s.job_id
WHERE j.title LIKE '%software%'
   OR e.name LIKE '%software%'
   OR j.location LIKE '%software%';
```

Sample Test Data (inserted into test database):
```
INSERT INTO Employer (name) VALUES ('Google'), ('Meta'), ('Amazon');
INSERT INTO JobPosting (employer_id, title, location, term)
VALUES
    (1, 'Software Engineer Intern', 'Waterloo, ON', 'Winter 2025'),
    (2, 'Data Analyst Co-op', 'Toronto, ON', 'Spring 2025'),
    (3, 'Software Developer', 'Vancouver, BC', 'Fall 2025');
INSERT INTO Salary (job_id, hourly_rate, hours_per_week)
VALUES (1, 40.00, 40), (2, 35.00, 37), (3, 42.00, 40);
```

Expected Output (test-sample.out):

title	employer	location	term	hourly_rate
Software Engineer Intern	Google	Waterloo, ON	Winter 2025	40.00
Software Developer	Amazon	Vancouver, BC	Fall 2025	42.00

R6c. SQL Query, Testing with Production Data

Indexing to boost speed on lookup time

```
CREATE INDEX idx_job_title ON JobPosting(title);
CREATE INDEX idx_employer_name ON Employer(name);
CREATE INDEX idx_job_location ON JobPosting(location);
```

Run the following before and after indexing to see changes and improvements in the processing pathway

```
EXPLAIN SELECT j.title, e.name, j.location, s.hourly_rate
FROM JobPosting j
JOIN Employer e ON j.employer_id = e.employer_id
JOIN Salary s ON j.job_id = s.job_id
WHERE j.title LIKE '%software%';
```

R6d. Implementation, Snapshot, Testing

Flask Interface Snapshot Description:

- A search bar labeled “Search Jobs” on /search.
- When a keyword is entered, a new results page displays a table with matched job entries.

How to Test:

- Run the Flask app and go to /search?q=software.
- The server executes the SQL query with the parameter %software%.
- Verify that only relevant job postings appear.
- Compare the displayed results to the expected test-sample.out.

User View Example in Browser:

```
---------------------------------------------------------------------------------------------
| Job Title                    | Employer      | Location              | Pay   |
---------------------------------------------------------------------------------------------
| Software Engineer Intern     | Google        | Waterloo, ON          | 40.00 |
| Software Developer           | Amazon        | Vancouver, BC         | 42.00 |
---------------------------------------------------------------------------------------------
```

R7a. Feature Interface Design
Feature Name: Top-Paying Companies by Given Role
User: General user (student or visitor)

Description:
Users can select a specific job role (e.g. Software Engineer Intern) and view the companies with that job role ranked by average hourly salary. This allows students to identify employers by their average salary and gauge the competitiveness of positions they may consider applying for.

User Interaction Flow:

1. The user navigates to the “Top Companies by Role” page.
2. The page will display a dropdown menu with a list of job role titles that the user can select from.
3. After selecting the job role title the Flask app executes a SQL query to retrieve the top 20 companies by average hourly rate within the selected job role.
4. The results shall be displayed in a table format in descending order based on hourly rate.

INTERNAL

```
@app.route("/top-companies", methods=['GET', 'POST'])
def top_companies():
    conn = get_db()
    cursor = conn.cursor(dictionary=True)
    role = request.form.get('role') if request.method == 'POST' else None
    if role:
        cursor.execute("""
            SELECT e.name AS company,
            ROUND(AVG(s.hourly_rate), 2) AS avg_hourly,
            COUNT(*) AS n_reports
            FROM Employer e
            JOIN JobPosting j ON j.employer_id = e.employer_id
            JOIN Salary s ON s.job_id = j.job_id
            WHERE j.title = %s
            GROUP BY e.name
            ORDER BY avg_hourly DESC
            LIMIT 20;
        """, (role,))
        rows = cursor.fetchall()
        conn.close()
        return render_template("top_companies.html", rows=rows, role=role)
    else:
        cursor.execute("SELECT DISTINCT title FROM JobPosting;")
        roles = [row['title'] for row in cursor.fetchall()]
        conn.close()
        return render_template("select_role.html", roles=roles)
```

R7b. SQL Query, Testing with Sample Data

SQL Query:

```
SELECT e.name AS company,
ROUND(AVG(s.hourly_rate), 2) AS avg_hourly,
COUNT(*) AS n_reports
FROM Employer e
JOIN JobPosting j ON j.employer_id = e.employer_id
JOIN Salary s ON s.job_id = j.job_id
WHERE j.title = %s
GROUP BY e.name
ORDER BY avg_hourly DESC
LIMIT 20;
```

Sample Test Data (inserted into test database):

```
INSERT INTO Employer (name) VALUES ('Microsoft'), ('Nvidia'), ('Apple');
INSERT INTO JobPosting (employer_id, title, location, term)
VALUES
    (1, 'Software Engineer Intern', 'New York', 'Summer 2025'),
    (2, 'Software Engineer Intern', 'San Francisco', 'Fall 2025'),
    (3, 'Software Engineer Intern', 'Cupertino', 'Winter 2025'),
    (1, 'Software Engineer Intern', 'Chicago', 'Winter 2026');
INSERT INTO Salary (job_id, hourly_rate, hours_per_week)
VALUES
    (1, 84.00, 40),
    (2, 67.00, 40),
    (3, 42.00, 40),
    (4, 80.00, 40);
```

Expected Output (test-sample.out):

company	avg_hourly	n_reports
Microsoft	82.00	2
Nvidia	67.00	1
Apple	42.00	1

R7c. SQL Query, Testing with Production Data

(Will complete in M2)

R7d. Implementation, Snapshot, Testing

(Will complete in M2)

R8a. Feature Interface Design
Feature name: Average Salary by Faculty / Program
User: General user (student or visitor)

Description:
Users can quickly see average hourly co-op pay aggregated by faculty and program. Filters (faculty, program keyword, term) are optional.

User Interaction Flow:

1. The user opens "Salary Insights → By Faculty / Program"
2. A small filter panel appears with:
    - Dropdown: Faculty (e.g. Engineering, Math, Arts, etc)
    - Text input: Program contains (e.g. "Computer Science", "Statistics")
    - Dropdown: Term (e.g. Winter 2025)
    - Button: Show Averages
3. On submit, the Flask app runs a parameterised SQL query and renders a table:
    - Faculty | Program | Avg Hourly Rate | #Placements

INTERNAL
```
@app.route("/avg-salary", methods=["GET"])
def avg_salary_by_faculty_program():
    fac = request.args.get("faculty", "").strip()
    prog = request.args.get("program_kw", "").strip()
    term = request.args.get("term", "").strip()

    fac = f"%{fac}%" if fac else "%"
    prog = f"%{prog}%" if prog else "%"
    term = f"%{term}%" if term else "%"

    conn = get_db_connection()
    cur = conn.cursor(dictionary=True)
    cur.execute("""
        SELECT s.faculty,
            s.program,
            ROUND(AVG(sa.hourly_rate), 2) AS avg_hourly_rate,
            COUNT(*) AS n_placements
        FROM Placement p
        JOIN Student s ON p.student_id = s.student_id
        JOIN Salary sa ON p.job_id = sa.job_id
        JOIN JobPosting j ON p.job_id = j.job_id
        WHERE s.faculty LIKE %s
            AND s.program LIKE %s
            AND j.term LIKE %s
        GROUP BY s.faculty, s.program
        ORDER BY avg_hourly_rate DESC, n_placements DESC;
     """, (fac, prog, term))
    rows = cur.fetchall()
    cur.close(); conn.close()
    return render_template("avg_salary.html", rows=rows)
```

R8b. SQL Query, Testing with Sample Data

SQL Query:

```
SELECT s.faculty, s.program,
      ROUND(AVG(sa.hourly_rate), 2) AS avg_hourly_rate,
      COUNT(*) AS n_placements
FROM Placement p
JOIN Student s ON p.student_id = s.student_id
JOIN Salary sa ON p.job_id = sa.job_id
JOIN JobPosting j ON p.job_id = j.job_id
WHERE s.faculty = 'Engineering'
      AND s.program LIKE '%Computer Science%'
      AND j.term = 'Winter 2025'
GROUP BY s.faculty, s.program
ORDER BY avg_hourly_rate DESC, n_placements DESC;
```

Sample Test Data (inserted into test database):

```
– Employers
INSERT INTO Employer (employer_id, name) VALUES
(1, 'Google'), (2, 'Meta'), (3, 'Amazon');

– Students
INSERT INTO Student (student_id, faculty, program) VALUES
(101, 'Engineering', 'Computer Science'),
(102, 'Engineering', 'Software Engineering'),
(103, 'Math', 'Statistics');

– Jobs
INSERT INTO JobPosting (job_id, employer_id, title, location, term) VALUES
(201, 1, 'Software Engineer Intern', 'Waterloo, ON', 'Winter 2025'),
(202, 2, 'Data Analyst Co-op', 'Toronto, ON', 'Spring 2025'),
(203, 3, 'Software Developer', 'Vancouver, BC', 'Fall 2025'),
(204, 1, 'SWE Intern (Backend)', 'Waterloo, ON', 'Winter 2025');

-- Salaries
INSERT INTO Salary (job_id, hourly_rate, hours_per_week) VALUES
(201, 40.00, 40), (202, 35.00, 37), (203, 42.00, 40), (204, 46.00, 40);

-- Placements (who worked where)
INSERT INTO Placement (student_id, job_id) VALUES
(101, 201),  -- Eng/CS at Google Winter 2025 ($40)
(101, 204),  -- Eng/CS at Google Winter 2025 ($46)
(102, 203),  -- Eng/SE at Amazon Fall 2025 ($42)
(103, 202);  -- Math/Statistics at Meta Spring 2025 ($35)
```

Expected Output (test-sample.out):

faculty	program	avg_hourly_rate	n_placements
Math	Computer Science	43.00	2

Another Expected Output (no filters: show all):

faculty	program	avg_hourly_rate	n_placements
Engineering	Electrical Engineering	42.00	1

Math	Computer Science	43.00	2
Math	Statistics	35.00	1

R8c. SQL Query, Testing with Production Data

Helpful Indexes:

```
CREATE INDEX idx_student_faculty ON Student(faculty);
CREATE INDEX idx_student_program ON Student(program);
CREATE INDEX idx_job_term ON JobPosting(term);
CREATE INDEX idx_salary_job ON Salary(job_id);
CREATE INDEX idx_place_student ON Placement(student_id);
CREATE INDEX idx_place_job ON Placement(job_id);
```

Explain (before / after indexes) to observe plan changes:

```
EXPLAIN
SELECT s.faculty, s.program,
        ROUND(AVG(sa.hourly_rate), 2) AS avg_hourly_rate,
        COUNT(*)
CREATE INDEX idx_student_program ON Student(program);
CREATE INDEX idx_job_term ON JobPosting(term);
CREATE INDEX idx_salary_job ON Salary(job_id);
CREATE INDEX idx_place_student ON Placement(student_id);
CREATE INDEX idx_place_job ON Placement(job_id);
```

R8d. Implementation, Snapshot, Testing

Flask Interface Snapshot (description)

- Route: /avg-salary
- Filter: Faculty (dropdown), Program contains (text), Term (dropdown)
- Table columns: Faculty | Program | Avg Hourly Rate | #Placements
- Sorting by Avg Hourly Rate (desc), ties by #Placements (desc)

How to Test:

1. Load the sample data above into a test
2. Run the Flask app and open:
    - /avg-salary?faculty=Engineering&program_kw=Computer%20Science&term=Winter%202025
3. Verify the table matches the Expected Output
4. Try broader queries (only faculty, only term, empty filters) and confirm grouping / averages

User View Example in Browser:

Faculty	**Program**	**Avg Hourly Rate**	**#Placements**
Engineering	Electrical Engineering	42.00	1
Math	Computer Science	43.00	2
Math	Statistics	35.00	1

R9a. Feature Interface Design
Feature Name: Average Hourly Salary by Term
User: General user (student or visitor)

Description:
Users can view the average hourly wage for each academic term (e.g., *Winter 2025*, *Fall 2025*). This allows them to quickly compare how co-op pay levels change across terms and seasons.

User Interaction Flow:

1. The user navigates to the "Average by Term" page.

2. The page displays a table with the following columns: Term, Average Hourly Wage ($/hr), and Number of Reports.

3. When the user visits this page, the Flask app executes an SQL aggregation query that groups all salaries by term.

4. The results are displayed in a table format sorted by term order.

INTERNAL

```
@app.route("/avg-by-term")
def avg_by_term():
    conn = get_db()
    cursor = conn.cursor(dictionary=True)
    cursor.execute("""
        SELECT j.term,
               ROUND(AVG(s.hourly_rate), 2) AS avg_hourly,
               COUNT(*) AS n_reports
        FROM JobPosting j
        JOIN Salary s ON s.job_id = j.job_id
        GROUP BY j.term
        ORDER BY j.term;
    """)
    rows = cursor.fetchall()
    conn.close()
    return render_template("avg_by_term.html", rows=rows)
```

R9b. SQL Query, Testing with Sample Data

SQL Query:

```
SELECT j.term,
       ROUND(AVG(s.hourly_rate), 2) AS avg_hourly,
       COUNT(*) AS n_reports
FROM JobPosting j
JOIN Salary s ON s.job_id = j.job_id
GROUP BY j.term
ORDER BY j.term;
```

Sample Test Data (inserted into test database):

```
INSERT INTO Employer (name) VALUES ('Google'), ('Shopify'), ('TinyStart');
INSERT INTO JobPosting (employer_id, title, location, term)
VALUES
    (1, 'SWE Intern', 'Waterloo', 'Winter 2025'),
    (2, 'Data Intern', 'Toronto', 'Fall 2025'),
    (3, 'Web Intern', 'Kitchener', 'Winter 2025'),
    (1, 'SWE Intern', 'Waterloo', 'Winter 2026');
INSERT INTO Salary (job_id, hourly_rate, hours_per_week)
VALUES
    (1, 45.00, 40),
    (2, 35.00, 40),
    (3, 16.00, 40),
    (4, 52.00, 40);
```

Expected Output (test-sample.out):

term	avg_hourly	n_reports
Fall 2025	35.00	1
Winter 2025	30.50	2
Winter 2026	52.00	1

R9c. SQL Query, Testing with Production Data

To improve performance for large datasets, indexes can be added on key columns used in grouping or filtering:

```
CREATE INDEX idx_job_term ON JobPosting(term);
CREATE INDEX idx_salary_job ON Salary(job_id);
```

To evaluate improvements, use:

```
EXPLAIN SELECT j.term,
               ROUND(AVG(s.hourly_rate), 2),
               COUNT(*)
FROM JobPosting j
JOIN Salary s ON s.job_id = j.job_id
GROUP BY j.term;
```

Compare query execution times before and after creating indexes.

R9d. Implementation, Snapshot, Testing

Flask Interface Snapshot Description:
The `/avg-by-term` page shows a table with each term's average hourly wage and number of salary reports. It automatically computes results on page load — no user input required.

How to Test:

1. Run the Flask app.
2. Go to `/avg-by-term`.
3. Verify that the table matches the expected averages from your `test-sample.sql`.
4. Compare displayed averages with manually calculated averages from the dataset.

User View Example in Browser:

```
--------------------------------------------------------------------
| Term          | Average Hourly ($/hr) | Reports |
--------------------------------------------------------------------
| Fall 2025     | 35.00                 | 1       |
| Winter 2025   | 30.50                 | 2       |
| Winter 2026   | 52.00                 | 1       |
--------------------------------------------------------------------
```