Milestone 3 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 synthetically generate more data to obtain a large enough production dataset to demonstrate the improvement in performance for queries and have a large table output. The dataset from Reddit contains companies that offer at least one job posting on WaterlooWorks along with their hourly salary, work term, and additional information such as benefits, year of job posting, and any companies put on a separate blacklist. However, as an example, the hourly salary for a company in this dataset is likely to be an aggregation of the individual job positions/roles from each employer, the columns/attributes will have differing names, and many records within the dataset will be empty. Therefore, we will transform the datasets (in CSV format) into a fixed and compatible relational format that can be loaded into our database. We (the project group) will be the administrators of the database system and will maintain the performance and correctness of the database, ensure backup and recovery, handle blacklisted companies (via procedures and triggers), etc. The core 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. In addition to these functionalities, the system also incorporates several advanced features that make use of more complex SQL capabilities. These include full-text relevance search, transactional blacklist entry management, percentile-based salary analysis, blacklist-aware safe employer recommendations, and enhanced views of historical blacklist activity, etc. Together, these features provide students with a more comprehensive and insightful way to explore co-op salary data.

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, transactions, 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. We will use HTML on top of Flask as the lightweight interface for the frontend of our application; it will have interactive elements such as dropdown menus, buttons, textboxes, etc. Finally, we ran the MySQL database on the Docker platform because running *sudo* on our machines did not work and the Docker container offered an operating-system-level virtualization that allowed MySQL to run in a properly setup (and non-corrupted) environment isolated from the rest of the machine.

The sample data used to populate our database is sourced from publicly available datasets that have information related to internship, work, and co-op opportunities. Our primary sources for our datasets are the self-reported co-op salaries datasheet from WaterlooWorks co-op postings, Kaggle datasets on internships, and synthetically generating more data. We will select a small subset of records from these datasets (around 30-50 records) for initial testing and format them to align with our database schema in the early stages. To populate our database with the sample data, we inserted records explicitly by calling INSERT INTO queries in a .sql file because it was far easier to do and the data transformation script was more complicated and better suited for processing the final production dataset.

Our production dataset will be the same Waterloo Co-op Salaries + Blacklist and Kaggle Internship Opportunities publicly available datasets; these combined with the synthetically generated data gives our production dataset > 10,000 records. Our production dataset in CSV (and DataFrame) format will be processed through a Python data transformation script that standardizes the column names, converts any monthly salaries into equivalent hourly rates, cleans and removes duplicate entries, maps the reasons for blacklists, and fills in missing fields/attributes with realistic values. The script will use *Pandas* and *SQLAlchemy* libraries to load the data into the database tables and help resolve any foreign key dependencies between tables, *re* library to handle regular expressions in the data, and *Faker* library to generate synthetic data.

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

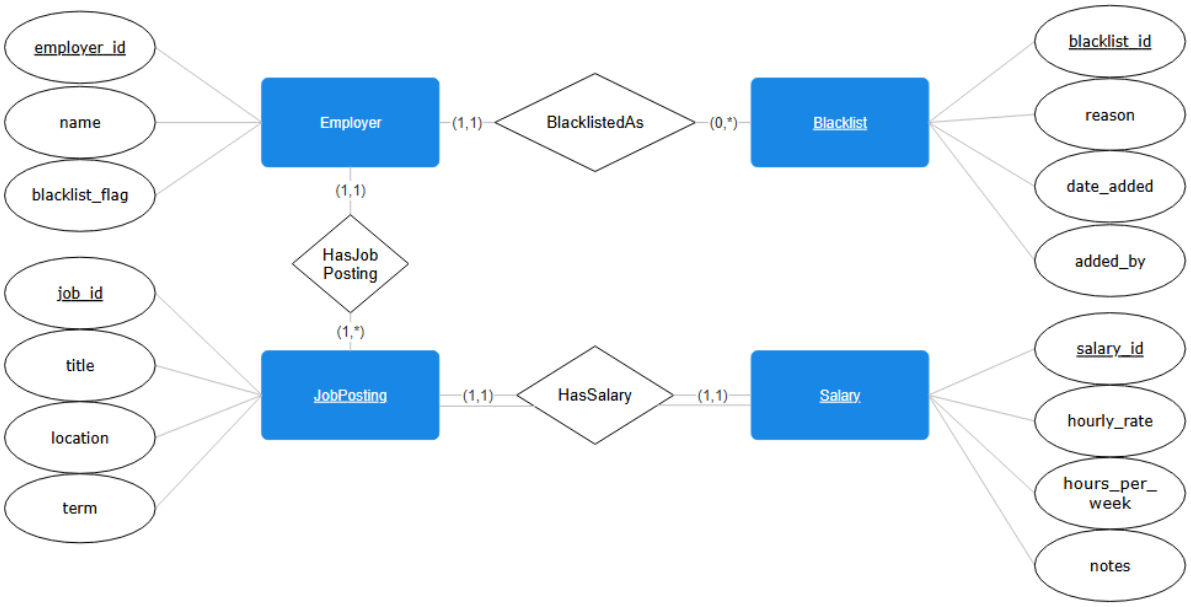
- Vincent Chen: Worked on designing the database schema, E/R diagram, all of R5, and wrote SQL files for creating tables, constraints, and triggers, etc (C2), and R6. Added more features in application code, updated SQL files to reflect any schema changes (C2), and worked on R10, R11, R12.
- Eunseo Lee: Worked on test-sample.sql file for Task 5 and the test-sample.out for testing our sample database (C3) and R8.
  Updated test-sample.sql and test.sample.out files to include all 5 basic features (C3), added more features in application code, refined all the sections from M1 to reflect project's progress, and worked on R13.
- 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.
  Worked on Production Data Plan (R4), added more features in application code, updated README file, and worked on R15.
- Andrew Lu: Worked on the Milestone 1 Report (R1, R2, R3) and R7.
  Updated README file, worked on Production Data Plan (R4), test-production.sql & test-production.out files (C4), and R14.

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



R5c. 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

We will use indexes on relevant columns to speed up the retrieval of companies by highest-paying salary:

```
CREATE INDEX idx_job_title ON JobPosting(title);
CREATE INDEX idx_job_employer ON JobPosting(employer_id);
CREATE INDEX idx_salary_job ON Salary(job_id);
```

We will use the EXPLAIN statement to analyze the execution plan of the query without actually running it; we can compare query execution times before and after creating indexes.

```
EXPLAIN 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 = 'Software Engineer Intern'
GROUP BY e.name
ORDER BY avg_hourly DESC
LIMIT 20;
```

R7d. Implementation, Snapshot, Testing

Flask Interface Snapshot Description:
The `/top-companies` page includes a form for selecting a role/position and displays a table of companies with the highest hourly salary and number of targeted roles/positions offered at that company (reports). It computes the results dynamically on user input since it requires an input role/position to be entered by the user.
How to Test:

1. Run the Flask app.
2. Go to `/top-companies` page.
3. Verify the table matches the expected output from your `test-sample.sql`.
4. Compare displayed averages with manually calculated averages from the dataset.

User View Example in Browser:

```
-------------------------------------------------------------------
| Company      | Average Hourly ($/hr) | Reports |
-------------------------------------------------------------------
| Microsoft    | 82.00                 | 2       |
| Nvidia       | 67.00                 | 1       |
| Apple        | 42.00                 | 1       |
-------------------------------------------------------------------
```

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       |
-------------------------------------------------------------------
```

R10a. Feature Interface Design
Feature Name: View Blacklisted Employers
User: General user (student or visitor)

Description:
Users can view a list of employers that have been 'blacklisted' by the system. The list displays the employer's name, the reason for the blacklist entry, and the date the entry was added. This feature allows students to be aware of employers with known issues, such as poor working conditions or pay disputes.

User Interaction Flow:

1. The user navigates to the "Employer Blacklist" page.
2. The page automatically displays a table with all employers that have an active blacklist_flag and their corresponding reasons from the Blacklist table.
3. When the user visits this page, the Flask app executes an SQL query joining the Employer and Blacklist tables.
4. The results are displayed in a table, sorted by the date the reason was added.

INTERNAL

```
@app.route("/blacklist")
def view_blacklist():
    conn = get_db_connection()
    cursor = conn.cursor(dictionary=True)
    cursor.execute("""
        SELECT e.name AS employer_name,
               b.reason,
               b.date_added
        FROM Employer e
        JOIN Blacklist b ON e.employer_id = b.employer_id
        WHERE e.blacklist_flag = TRUE
        ORDER BY b.date_added DESC, e.name;
    """)
    rows = cursor.fetchall()
    conn.close()
    return render_template("blacklist.html", reports=rows)
```

R10b. SQL Query, Testing with Sample Data
SQL Query:

```
SELECT e.name AS employer_name,
       b.reason,
       b.date_added
FROM Employer e
JOIN Blacklist b ON e.employer_id = b.employer_id
WHERE e.blacklist_flag = TRUE
ORDER BY b.date_added DESC, e.name;
```

Sample Test Data (inserted into test database):

```
INSERT INTO Employer (employer_id, name, blacklist_flag)
VALUES
    (1, 'Good Company', FALSE),
    (2, 'Risky Startup', TRUE),
    (3, 'Scam Co', TRUE);
```

```
INSERT INTO Blacklist (employer_id, reason, date_added, added_by)
VALUES
    (2, 'Repeatedly failed to pay students', '2025-01-15', 'admin'),
    (3, 'Toxic work environment reported', '2025-02-01', 'admin'),
    (3, 'Rescinded offers last minute', '2025-02-10', 'admin');
```

Expected Output (test-sample.out):

"employer_name"	, "reason"	, "date_added"
"Scam Co"	, "Rescinded offers last minute"	, "2025-02-10"
"Scam Co"	, "Toxic work environment reported"	, "2025-02-01"
"Risky Startup"	, "Repeatedly failed to pay students"	, "2025-01-15"

R10c. SQL Query, Testing with Production Data

Indexing to boost speed on lookup time:

```
CREATE INDEX idx_employer_blacklist_flag ON Employer(blacklist_flag);
CREATE INDEX idx_blacklist_employer_id ON Blacklist(employer_id);
```

Run the following before and after indexing to see changes:

```
EXPLAIN SELECT e.name, b.reason, b.date_added
FROM Employer e
JOIN Blacklist b ON e.employer_id = b.employer_id
WHERE e.blacklist_flag = TRUE;
```

R10d. Implementation, Snapshot, Testing

Flask Interface Snapshot Description:

- The /blacklist page shows a table with columns: "Employer", "Reason", and "Date Reported".
- The page is read-only for general users, as blacklist management is an admin function.

How to Test:

- Run the Flask app and go to /blacklist.
- Verify that the table displays all entries from the Blacklist table where the corresponding Employer has blacklist_flag = TRUE.
- Verify that "Good Company" (which is not flagged) does not appear in the list.
- Compare the displayed results to the test-sample.out.

User View Example in Browser:

Employer	Reason	Date Reported
Scam Co	Rescinded offers last minute	2025-02-10
Scam Co	Toxic work environment reported	2025-02-01
Risky Startup	Repeatedly failed to pay students	2025-01-15

R11a. Feature Interface Design
Feature Name: Full-Text Relevance Search
User: General user (student or visitor)

Description:
Users can search for co-op jobs by typing keywords into a search bar, similar to R6. The keywords are matched against the job title, employer name, and location fields. Unlike R6, the results are ranked by relevance, with matches in the title weighted more heavily than matches in other fields. This is considered more advanced due to its query complexity (using MATCH...AGAINST and relevance scoring).

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 one or more keywords (e.g., "Software Engineer", "Toronto Data").
4. Upon clicking Search, the Flask app executes a parameterized SQL query using MATCH...AGAINST to find and rank relevant jobs.
5. The results are displayed in a table, sorted from most relevant to least relevant.

INTERNAL

```
@app.route("/advanced-search", methods=["GET"])
def advanced_search():
    keyword = request.args.get("q", "")
    conn = get_db_connection()
    cursor = conn.cursor(dictionary=True)

    # This query uses FTS and ranks results by relevance
    cursor.execute("""
        SELECT j.title, e.name AS employer, j.location, s.hourly_rate,
               MATCH(j.title) AGAINST(%s IN NATURAL LANGUAGE MODE) AS title_relevance,
               MATCH(e.name, j.location) AGAINST(%s IN NATURAL LANGUAGE MODE) AS
other_relevance
        FROM JobPosting j
        JOIN Employer e ON j.employer_id = e.employer_id
        JOIN Salary s ON j.job_id = s.job_id
        WHERE MATCH(j.title, e.name, j.location) AGAINST(%s IN NATURAL LANGUAGE MODE)
        ORDER BY (title_relevance * 2.0) + other_relevance DESC;
    """, (keyword, keyword, keyword))

    rows = cursor.fetchall()
    conn.close()
    return render_template("search_results.html", jobs=rows)
```

R11b. SQL Query, Testing with Sample Data
SQL Query (for testing 'Software'):

```
SELECT j.title, e.name,
       MATCH(j.title) AGAINST('Software') AS title_relevance
FROM JobPosting j
JOIN Employer e ON j.employer_id = e.employer_id
WHERE MATCH(j.title, e.name, j.location) AGAINST('Software')
ORDER BY title_relevance DESC;
```

Sample Test Data:

```
INSERT INTO Employer (name) VALUES ('Acme Software'), ('Data Corp'), ('Tech Solutions');
INSERT INTO JobPosting (employer_id, title, location, term)
VALUES
    (1, 'Junior Developer', 'Toronto', 'Winter 2025'), -- Employer match
    (2, 'Software Engineer Intern', 'Waterloo', 'Winter 2025'), -- Title match (strong)
    (3, 'Analyst (uses software)', 'Toronto', 'Spring 2025'); -- Title match (weak)
INSERT INTO Salary (job_id, hourly_rate) VALUES (1, 30.00), (2, 45.00), (3, 32.00);
```

Expected Output (test-sample.out for 'Software'):

Note: The LIKE query from R6 would return all three, but FTS (especially with ranking) will correctly prioritize the job with "Software" in the title.

"title"	, "employer"	, ...
"Software Engineer Intern"	, "Data Corp"	, ...
"Junior Developer"	, "Acme Software"	, ...
"Analyst (uses software)"	, "Tech Solutions"	, …

R11c. Use of Advanced MySQL Features

This feature is advanced because it uses FULLTEXT indices and relevance-based queries (MATCH...AGAINST), which is a specific requirement for this milestone.

Advanced MySQL Feature: FULLTEXT indexing is used, which is more specialized than standard B-tree indexes .

```
CREATE FULLTEXT INDEX idx_fts_job_title ON JobPosting(title);
CREATE FULLTEXT INDEX idx_fts_employer_loc ON Employer(name, location);
-- Note: This is an example; schema in r5 shows location on JobPosting [cite: 200]
-- Correct FTS index based on r5/r6 schema:
CREATE FULLTEXT INDEX idx_fts_jobs ON JobPosting(title, location);
CREATE FULLTEXT INDEX idx_fts_employer ON Employer(name);
```

Query Complexity: The query uses MATCH...AGAINST instead of LIKE. It also introduces calculated relevance scoring (ORDER BY (title_relevance * 2.0) + ...) to provide more useful, ranked results, which is significantly more complex than the simple OR logic in R6.

R11d. Implementation, Snapshot, Testing

Flask Interface Snapshot Description: The interface looks identical to R6 (a search bar), but the results returned are more relevant and are not in a random or simple alphabetical order.

How to Test:

- Load the sample data and create the FULLTEXT indices.
- Run the Flask app and go to /advanced-search?q=Software.
- Verify that "Software Engineer Intern" is ranked higher than "Acme Software" (employer match) or "Analyst (uses software)" (weak title match).
- Compare this to the R6 query (/search?q=software), which would return all three results with no meaningful order.

R12a. Feature Interface Design
Feature Name: Add Blacklist Entry (Transactional)
User: Admin user (not a general student/visitor)

Description:
An admin user can blacklist an employer by providing a reason. To maintain data integrity, this action must be transactional. The system must (1) INSERT a new record into the Blacklist table and (2) UPDATE the Employer table to set blacklist_flag = TRUE. If either operation fails, the entire transaction is rolled back, preventing a-state where an employer is flagged without a reason, or has a reason record but is not flagged.

User Interaction Flow:

1. The admin navigates to an "/admin/blacklist-add" page.
2. A form is displayed with a dropdown of employers and a text area for "Reason".
3. The admin selects an employer (e.g., "Scam Co"), enters a reason, and clicks "Submit".
4. The Flask app executes two SQL commands inside a single transaction.
5. If successful, the admin is redirected. If it fails, an error is shown and no data is changed.

INTERNAL

```
@app.route("/admin/blacklist-add", methods=["POST"])
def add_blacklist_entry():
    # (Assume admin is authenticated)
    employer_id = request.form.get("employer_id")
    reason = request.form.get("reason")
    admin_user = "admin@system.com" # From session

    conn = get_db_connection()
    cursor = conn.cursor()

    try:
        # Use an explicit transaction
        conn.start_transaction()

        # 1. Insert the detailed reason into the Blacklist table
        cursor.execute("""
            INSERT INTO Blacklist (employer_id, reason, date_added, added_by)
            VALUES (%s, %s, CURDATE(), %s)
        """, (employer_id, reason, admin_user))

        # 2. Update the flag on the Employer table
        cursor.execute("""
            UPDATE Employer SET blacklist_flag = TRUE
            WHERE employer_id = %s
        """, (employer_id,))

        # If both succeed, commit the changes
        conn.commit()

    except mysql.connector.Error as err:
        # If anything fails, roll back all changes
        conn.rollback()
```

```
        return "Error: Could not add blacklist entry. Data has been rolled back.", 500
    finally:
        conn.close()

    return "Employer blacklisted successfully.", 200
```

R12b. SQL Query, Testing with Sample Data

SQL Queries (executed transactionally):

```
INSERT INTO Blacklist (employer_id, reason, ...) VALUES (1, 'Failed to pay', ...);
UPDATE Employer SET blacklist_flag = TRUE WHERE employer_id = 1;
```

Sample Test Data (Initial State):

```
INSERT INTO Employer (employer_id, name, blacklist_flag)
VALUES (1, 'Risky Startup', FALSE);
Expected Output (Final State in DB):
```

Employer table: (1, 'Risky Startup', TRUE)

Blacklist table: (..., 1, 'Failed to pay', ...)

R12c. Use of Advanced MySQL Features

This feature is advanced because it uses Transactions to guarantee data integrity (atomicity), which is a core advanced database concept.

Advanced MySQL Feature: The conn.start_transaction(), conn.commit(), and conn.rollback() methods are used to wrap the two SQL commands.

Purpose: This directly addresses the schema's requirement to keep the Employer flag and Blacklist table synchronized. Without a transaction, it would be possible for the INSERT to succeed but the UPDATE to fail (e.g., due to a brief connection loss), leaving the database in an inconsistent state. This feature correctly prevents that.

R12d. Implementation, Snapshot, Testing

Flask Interface Snapshot Description: A simple admin form at /admin/blacklist-add with an employer dropdown and a "Reason" text box.

How to Test:

Test 1 (Success): Load the sample data. Call the /admin/blacklist-add endpoint with employer_id=1 and a reason. Verify that both the Blacklist table has the new row and the Employer table's blacklist_flag is set to TRUE.

Test 2 (Failure/Rollback): Modify the code to force the UPDATE query to fail (e.g., UPDATE non_existent_table ...). Call the endpoint. Verify that the conn.rollback() is triggered and no new row is added to the Blacklist table. The database state should be unchanged from the initial state.

R13a. Feature Interface Design

Feature Name: Safe Employer Recommendations by Faculty (Blacklist-Aware + Low-Pay Filter)
User: General user (student or visitor)

Description:
This feature recommends “safe” employers for a given faculty by looking at historical co-op placements and filtering out:

1. Employers that are blacklisted, and
2. Employers that often pay significantly below the average for that faculty.

For a chosen faculty (e.g. Engineering), the system:

- Computes the average hourly rate for that faculty across all placements.
- Computes, for each employer who has hired that faculty:
    - Their average hourly rate for that faculty.
    - The number of placements.
- Excludes employers who:
    - Are blacklisted (blacklist_flag = TRUE), or
    - Have any placement for that faculty with pay < 70% of the faculty average (using a correlated NOT EXISTS subquery).
- Returns a ranked list of employers that:
    - Are not blacklisted
    - Have at least 2 placements for that faculty
    - Never paid that faculty “too low” compared with the global faculty average.

User Interaction Flow:

1. The user opens “Salary Insights → Safe Employers by Faculty” (/safe-employers).
2. A small filter panel appears with: Dropdown: Faculty (e.g. Engineering, Math, Arts)
3. The user selects a faculty and clicks “Show Recommendations.”
4. The Flask app runs a CTE-based, multi-join, NOT EXISTS SQL query.
5. The results are shown in a table:
   Employer | Faculty | Avg Hourly Rate | #Placements | Blacklisted?

INTERNAL

```
@app.route("/safe-employers")
def safe_employers():
    faculty = request.args.get("faculty", "").strip()

    conn = get_db_connection()
    cur = conn.cursor(dictionary=True)

    query = """
      WITH faculty_avg AS (
        SELECT s.faculty, AVG(sa.hourly_rate) AS faculty_avg_rate
        FROM Placement p
        JOIN Student s ON p.student_id = s.student_id
        JOIN Salary sa ON sa.job_id = p.job_id
        GROUP BY s.faculty
      ),
```

```
      employer_faculty_stats AS (
        SELECT
          e.employer_id,
          e.name AS employer,
          s.faculty,
          AVG(sa.hourly_rate) AS employer_avg_rate,
          COUNT(*) AS n_placements
        FROM Placement p
        JOIN Student s ON p.student_id = s.student_id
        JOIN Salary sa ON sa.job_id = p.job_id
        JOIN JobPosting j ON j.job_id = p.job_id
        JOIN Employer e ON e.employer_id = j.employer_id
        GROUP BY e.employer_id, e.name, s.faculty
      )
      SELECT
        efs.employer,
        efs.faculty,
        ROUND(efs.employer_avg_rate, 2) AS avg_hourly_rate,
        efs.n_placements
      FROM employer_faculty_stats efs
      JOIN faculty_avg fa
        ON fa.faculty = efs.faculty
      JOIN Employer e
        ON e.employer_id = efs.employer_id
      WHERE (%s = '' OR efs.faculty = %s)
        AND e.blacklist_flag = FALSE
        AND efs.n_placements >= 2
        AND NOT EXISTS (
          SELECT 1
          FROM Placement p2
          JOIN Student s2 ON p2.student_id = s2.student_id
          JOIN Salary sa2 ON sa2.job_id = p2.job_id
          JOIN JobPosting j2 ON j2.job_id = p2.job_id
          WHERE s2.faculty = efs.faculty
            AND j2.employer_id = efs.employer_id
            AND sa2.hourly_rate < 0.7 * fa.faculty_avg_rate
        )
      ORDER BY avg_hourly_rate DESC, n_placements DESC;
    """

    cur.execute(query, (faculty, faculty))
    rows = cur.fetchall()
    cur.close(); conn.close()

    return render_template("safe_employers.html",
                           rows=rows, faculty=faculty)
```

R13b. SQL Query, Testing with Sample Data

SQL Query:

```
WITH faculty_avg AS (
    SELECT
        s.faculty,
```

```
        AVG(sa.hourly_rate) AS faculty_avg_rate
    FROM Placement p
    JOIN Student s ON p.student_id = s.student_id
    JOIN Salary sa ON sa.job_id = p.job_id
    GROUP BY s.faculty
),
employer_faculty_stats AS (
    SELECT
        e.employer_id,
        e.name AS employer,
        s.faculty,
        AVG(sa.hourly_rate) AS employer_avg_rate,
        COUNT(*) AS n_placements
    FROM Placement p
    JOIN Student s ON p.student_id = s.student_id
    JOIN Salary sa ON sa.job_id = p.job_id
    JOIN JobPosting j ON j.job_id = p.job_id
    JOIN Employer e ON e.employer_id = j.employer_id
    GROUP BY e.employer_id, e.name, s.faculty
)
SELECT
    efs.employer,
    efs.faculty,
    ROUND(efs.employer_avg_rate, 2) AS avg_hourly_rate,
    efs.n_placements
FROM employer_faculty_stats efs
JOIN faculty_avg fa
    ON fa.faculty = efs.faculty
JOIN Employer e
    ON e.employer_id = efs.employer_id
WHERE efs.faculty = 'Engineering'
  AND e.blacklist_flag = FALSE
  AND efs.n_placements >= 2
  AND NOT EXISTS (
      SELECT 1
      FROM Placement p2
      JOIN Student s2 ON p2.student_id = s2.student_id
      JOIN Salary sa2 ON sa2.job_id = p2.job_id
      JOIN JobPosting j2 ON j2.job_id = p2.job_id
      WHERE s2.faculty = efs.faculty
        AND j2.employer_id = efs.employer_id
        AND sa2.hourly_rate < 0.7 * fa.faculty_avg_rate
  )
ORDER BY avg_hourly_rate DESC, n_placements DESC;
```

Sample Test Data

```
-- Employers
INSERT INTO Employer (employer_id, name, blacklist_flag) VALUES
  (1, 'GoodPay Inc', FALSE),
  (2, 'LowPay Inc', FALSE),
  (3, 'Shady Corp', TRUE);

-- Students
```

```
INSERT INTO Student (student_id, faculty, program) VALUES
  (101, 'Engineering', 'Computer Science'),
  (102, 'Engineering', 'Electrical Engineering'),
  (103, 'Engineering', 'Mechanical Engineering'),
  (201, 'Math', ‘Statistics');

-- Job Postings
INSERT INTO JobPosting (job_id, employer_id, title, location, term)
  VALUES (301, 1, 'SWE Intern', 'Waterloo, ON', 'Winter 2025'),
  (302, 1, 'Backend Developer', 'Waterloo, ON', 'Spring 2025'),
  (303, 2, 'SWE Intern', 'Waterloo, ON', 'Winter 2025'),
  (304, 3, 'SWE Intern', 'Toronto, ON',  'Winter 2025'),
  (305, 1, 'Data Intern', 'Waterloo, ON', 'Fall 2025');

-- Salaries
INSERT INTO Salary (job_id, hourly_rate, hours_per_week) VALUES
  (301, 42.00, 40), -- GoodPay Inc, Eng
  (302, 45.00, 40), -- GoodPay Inc, Eng
  (303, 20.00, 40), -- LowPay Inc, Eng (very low)
  (304, 44.00, 40), -- Shady Corp, Eng
  (305, 44.00, 40); -- GoodPay Inc, Eng

-- Placements (who worked where) INSERT INTO Placement (student_id, job_id) VALUES
  (101, 301),  -- Eng @ GoodPay (42)
  (102, 302),  -- Eng @ GoodPay (45)
  (103, 305),  -- Eng @ GoodPay (44)
  (101, 303),  -- Eng @ LowPay (20)
  (102, 304);  -- Eng @ Shady (44)
```

R13c. Use of Advanced MySQL Features / Query Complexity

This feature is considered advanced because it uses non-trivial SQL patterns beyond basic joins and aggregates:

1. Common Table Expressions (CTEs) using WITH:
    - faculty_avg computes per-faculty average salary.
    - employer_faculty_stats computes per-employer, per-faculty averages and placement counts.
    - This separates logic into readable blocks and enables multi-stage aggregation.
2. Correlated NOT EXISTS subquery (anti-join):
    - The NOT EXISTS block checks, for each (employer, faculty) pair, whether there is any placement with hourly_rate < 0.7 * faculty_avg.
    - This is more advanced than simply using WHERE hourly_rate >= ... on a single row, because it excludes employers based on any low-paying record in their history.
3. Combination of multiple filters and group-level constraints:
    - Filtering by faculty.
    - Excluding blacklisted employers (blacklist_flag = FALSE).
    - Enforcing a minimum number of placements (n_placements >= 2).
    - Ordering by average hourly rate and volume.

R13d. Implementation, Snapshot, Testing

Flask Interface Snapshot Description:

- Route: /safe-employers
- Behaviour:
    - Only shows employers that:
        - Are not blacklisted
        - Have at least 2 placements from that faculty
        - Never paid < 70% of the faculty average

Sample Data Test (Engineering):

1. Load the sample data for part b.
2. Run the Flask app and go to /safe-employers?faculty=Engineering.
3. Verify that “GoodPay Inc” appears with its average hourly rate and 3 placements.
4. Confirm that “LowPay Inc” (very low pay) and “Shady Corp” (blacklisted) do not appear.

No Filter / Other Faculty Test:

1. Visit /safe-employers with no faculty parameter or with a different faculty value (e.g., faculty=Math).
2. Check that either:
    - Only safe employers for that faculty are shown, or
    - A “No safe employers found” message is displayed if there is no matching data.

R14a. Feature Interface Design

Feature Name: Auto-Flagging Low-Paying Employers

User: System (automatically triggered by the server/backend, not a general student/visitor)

Description:

When a new record is inserted or updated in the Salary table, a MySQL trigger will automatically calculate the average hourly rate across all jobs of the current employer. If this average falls below a set threshold rate (e.g. $20/hour), the trigger will set the blacklist_flag column of the employer in the Employer table to TRUE and insert a default reason into the Blacklist table. This operation will also make use of transactions to ensure data integrity and consistency in case of a server or power failure; partial operations already run will be rolled back to the state before the transaction was executed.

User Interaction Flow:

1. The admin navigates to the “/admin/add-salary” page.
2. A form is displayed with the fields:
    - Job ID (dropdown)
    - Hourly rate
    - Hours per week
    - Notes
3. The admin fills out the above fields and clicks “Submit”.
4. The Flask app inserts/updates the salary record in the Salary table. This SQL command will be inside the transaction.
5. The trigger will calculate the employer’s new average hourly rate across all their posted jobs; if this average is below the threshold rate, the blacklist_flag of the employer will be set to TRUE and a default reason is inserted into the Blacklist table. These SQL commands will also be inside the transaction.
6. If successful, the admin is redirected. If it fails, an error is shown and no data is changed.

INTERNAL

```
@app.route("/admin/add-salary", methods=["POST"])
def add_salary():
    # (Assume admin is authenticated)
    job_id = request.form.get("job_id")
    hourly_rate = float(request.form.get("hourly_rate"))
    hours_per_week = int(request.form.get("hours_per_week"))
    notes = request.form.get("notes")

    conn = get_db_connection()
    cursor = conn.cursor()

    try:
        conn.start_transaction()

        cursor.execute("""
            INSERT INTO Salary (job_id, hourly_rate, hours_per_week, notes)
            VALUES (%s, %s, %s, %s)
            ON DUPLICATE KEY UPDATE
            hourly_rate = VALUES(hourly_rate),
            hours_per_week = VALUES(hours_per_week),
```

```
            notes = VALUES(notes)
        """, (job_id, hourly_rate, hours_per_week, notes))

        conn.commit()

    except mysql.connector.Error as err:
        conn.rollback()
        return "Error: Could not insert/update salary. Data has been rolled back.", 500
    finally:
        conn.close()

    return “Salary added/updated successfully.”, 200
```

R14b. SQL Query, Testing with Sample Data

SQL Query:

```
-- No explicit TRANSACTION clause needed; trigger runs within the same transaction as the
INSERT into Salary table.

DELIMITER //
-- Create trigger on INSERT
CREATE TRIGGER auto_flag_low_pay AFTER INSERT ON Salary
FOR EACH ROW
BEGIN
DECLARE emp_id INT;
DECLARE avg_rate DECIMAL(6,2);

-- Get employer_id from JobPosting table
SELECT employer_id INTO emp_id
FROM JobPosting
WHERE job_id = NEW.job_id;

-- Compute the employer’s new average rate across all their jobs
SELECT AVG(hourly_rate) INTO avg_rate
FROM Salary s
JOIN JobPosting j ON s.job_id = j.job_id
WHERE j.employer_id = emp_id;

-- If new average rate < $20/hour, flag and add a blacklist entry
IF avg_rate < 20.00 THEN
UPDATE Employer
SET blacklist_flag = TRUE
WHERE employer_id = emp_id;

-- Insert default reason into Blacklist table
INSERT INTO Blacklist (employer_id, reason, date_added, added_by)
VALUES (emp_id, 'Auto-flagged for low average pay', CURDATE(), 'system');
END IF;

-- Create trigger on UPDATE (same logic as above trigger on INSERT)
CREATE TRIGGER auto_flag_low_pay_update AFTER UPDATE ON Salary
FOR EACH ROW
BEGIN
DECLARE emp_id INT;
DECLARE avg_rate DECIMAL(6,2);

-- Get employer_id from JobPosting table
SELECT employer_id INTO emp_id
```

```
FROM JobPosting
WHERE job_id = NEW.job_id;

-- Compute the employer's new average rate across all their jobs
SELECT AVG(hourly_rate) INTO avg_rate
FROM Salary s
JOIN JobPosting j ON s.job_id = j.job_id
WHERE j.employer_id = emp_id;

-- If new average rate < $20/hour, flag and add a blacklist entry
IF avg_rate < 20.00 THEN
UPDATE Employer
SET blacklist_flag = TRUE
WHERE employer_id = emp_id;

-- Insert default reason into Blacklist table
INSERT INTO Blacklist (employer_id, reason, date_added, added_by)
VALUES (emp_id, 'Auto-flagged for low average pay', CURDATE(), 'system');
END IF;

END;//
DELIMITER;
```

Sample Test Data:

```
INSERT INTO Employer (employer_id, name, blacklist_flag)
VALUES (1, 'LowPay Co', FALSE);

INSERT INTO JobPosting (employer_id, title, location, term)
VALUES (LAST_INSERT_ID(), 'Intern', 'Waterloo', 'Winter 2025');
INSERT INTO Salary (job_id, hourly_rate, hours_per_week)
VALUES (LAST_INSERT_ID(), 25.00, 40); -- average rate = 25.00 >= 20; trigger not called

INSERT INTO JobPosting (employer_id, title, location, term)
VALUES (1, 'Intern', 'Waterloo', 'Winter 2025');
INSERT INTO Salary (job_id, hourly_rate, hours_per_week)
VALUES (LAST_INSERT_ID(), 13.00, 40); -- average rate = 19.00 < 20; trigger called
```

Expected Output (test-sample.out):

Employer table:
(1, 'LowPay Co', TRUE)

JobPosting table:
(1, 1, 'Intern', 'Waterloo', 'Winter 2025')
(2, 1, 'Intern', 'Waterloo', 'Winter 2025')

Salary table:
(1, 1, 25.00, 40, NULL)
(2, 2, 13.00, 40, NULL)

Blacklist table:
(1, 1, 'Auto-flagged for low average pay', 2025-11-24, 'system')

R14c. Use of Advanced MySQL Features
This feature is advanced because it uses MySQL triggers, which are invoked automatically in response to events (INSERT/UPDATE ON Salary in this case) without needing an explicit call from the application. Triggers are able to execute logic such as aggregating averages and performing conditional inserts and updates across tables. Additionally MySQL transactions are needed to guarantee atomicity and data integrity in case of power or server failures; if a failure occurs during the execution of the first INSERT statement or trigger in a MySQL transaction, MySQL will raise an error and rollback everything already executed inside of `conn.start_transaction()` and `conn.commit()` back to the initial state. Triggers, transactions, and conditional inserts/updates based on AVG aggregation computation are considered advanced MySQL features and are more complex than basic features.

R14d. Implementation, Snapshot, Testing
Flask Interface Snapshot Description:
/admin/add-salary displays a form for the user to input salary details for a job. There is no explicit UI when blacklist entries are inserted or the trigger is run in the background.

How to Test:

- Insert high salary rates via the form or SQL inserts; verify blacklist_flag is FALSE.
- Insert low salary rates via the form or SQL; verify blacklist_flag is TRUE and a new entry is inserted in the Blacklist table.
- Failure and rollback: Temporarily add a UNIQUE constraint for Blacklist.reason attribute in the table schema to force a failure on INSERT; verify that the transaction rolls back and the values in every table in the database remain unchanged and no new records are inserted.
- Production: With large data, insert many varying salary rates from multiple employers; check that every employer with an average salary rate across all jobs < $20/hour has blacklist_flag = TRUE and they are inserted into the Blacklist table. All other employers should have blacklist_flag = FALSE and should not be inserted into the Blacklist table. Check that rollbacks revert the database to the initial state through temporary code that forces failures.

R15a. Feature Interface Design
Feature Name: Salary Percentiles & Bands
User: General user (student or visitor)

Description:
This feature lets users see where a given job's pay sits in the salary distribution for a specific role, city, and term. It uses window functions to compute percentiles and bands (e.g., deciles) across all postings that match the filters, and shows:

- Percentile of each job (e.g., 0.50 = median, 0.90 = top 10%)
- A band label (e.g., "Bottom 20%", "Middle 20%", "Top 10%")
- Aggregate stats like min, median, max, average hourly rate for the group

User Interaction Flow:

1. The user navigates to the "Salary Distribution" page (e.g., /salary-bands).
2. They see a form with filters:
    - Job Title (text input or dropdown, e.g., "Software Engineer Intern")
    - City (text input or dropdown, e.g., "Waterloo, ON")
    - Term (dropdown, e.g., "Winter 2026")
3. The user fills in one or more filters and clicks "View Distribution".
4. The system:
    - Filters the JobPosting + Salary tables to postings matching those criteria.
    - Uses window functions to compute the percentile and decile band for each posting within that group.
    - Computes group-level stats (min, median, max, mean).
5. The UI displays:
    - At the top: summary line like "For Software Engineer Intern in Waterloo, ON (Winter 2026), median hourly rate is $42.00, top 10% is $50.00+."
6. Then a table:

Employer	Title	Location	Hourly Rate	Percentile	Band
A Corp	SWE	Waterloo	40.00	0.30	Bottom 40%
B Corp	SWE	Waterloo	42.00	0.50	Middle 20%
C Corp	SWE	Waterloo	50.00	0.90	Top 10%

INTERNAL

```
@app.route("/salary-bands")
def salary_bands():
    title = request.args.get("title", "").strip()
    city  = request.args.get("city", "").strip()
    term  = request.args.get("term", "").strip()

    filters, params = [], []
    if title:
        filters.append("j.title = %s")
        params.append(title)
    if city:
        filters.append("j.location = %s")
        params.append(city)
    if term:
        filters.append("j.term = %s")
        params.append(term)

    where_clause = " AND ".join(filters) if filters else "1=1"

    query = f"""
      WITH filtered AS (
        SELECT j.title, j.location, j.term,
               e.name AS employer, 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 {where_clause} AND s.hourly_rate IS NOT NULL
      ),
      ranked AS (
        SELECT employer, title, location, term, hourly_rate,
               PERCENT_RANK() OVER (
                 PARTITION BY title, location, term
                 ORDER BY hourly_rate
               ) AS pct_rank,
               NTILE(10) OVER (
                 PARTITION BY title, location, term
                 ORDER BY hourly_rate
               ) AS decile
        FROM filtered
      )
      SELECT * FROM ranked ORDER BY hourly_rate;
    """

    conn = get_db_connection()
    cur = conn.cursor(dictionary=True)
    cur.execute(query, params)
    rows = cur.fetchall()
    conn.close()

    return render_template("salary_bands.html",
                           rows=rows, title=title, city=city, term=term)
```

R15b. SQL Query, Testing with Sample Data

SQL Query:

```
WITH filtered AS (
    SELECT
        j.title, j.location, j.term,
        e.name AS employer,
        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    = 'Software Engineer Intern'
      AND j.location = 'Waterloo, ON'
      AND j.term     = 'Winter 2026'
),
ranked AS (
    SELECT
        employer, title, location, term, hourly_rate,
        PERCENT_RANK() OVER (
            PARTITION BY title, location, term
            ORDER BY hourly_rate
        ) AS pct_rank,
        NTILE(10) OVER (
            PARTITION BY title, location, term
            ORDER BY hourly_rate
        ) AS decile
    FROM filtered
)
SELECT employer, title, location, term, hourly_rate,
       ROUND(pct_rank,2) AS pct_rank,
       decile
FROM ranked
ORDER BY hourly_rate;
```

Sample Test Data:

```
INSERT INTO Employer(name) VALUES ('A'),('B'),('C'),('D');
INSERT INTO JobPosting (employer_id, title, location, term) VALUES
  (1,'Software Engineer Intern','Waterloo, ON','Winter 2026'),
  (2,'Software Engineer Intern','Waterloo, ON','Winter 2026'),
  (3,'Software Engineer Intern','Waterloo, ON','Winter 2026'),
  (4,'Software Engineer Intern','Waterloo, ON','Winter 2026');
INSERT INTO Salary(job_id, hourly_rate) VALUES
  (1,30.00),(2,35.00),(3,40.00),(4,50.00);
```

Expected Output (test-sample.out for 'Software'):

```
employer  hourly_rate  pct_rank  decile
A         30.00        0.00      1
B         35.00        0.33      4
C         40.00        0.67      7
D         50.00        1.00      10
```

R15c. Use of Advanced MySQL Features
This feature is advanced because it uses window functions, which go beyond basic `GROUP BY` queries:

- `PERCENT_RANK()` computes each job's exact position in the salary distribution for the selected title, city, and term.
- `NTILE(10)` divides salaries into decile bands (e.g., top 10%, bottom 20%), something that cannot be done with standard aggregate functions.
- Both functions use `OVER (PARTITION BY … ORDER BY …)`, meaning calculations are performed within each group of postings rather than across the entire table.

These analytic window functions and partitioned computations are considered advanced SQL functionality and clearly exceed the complexity of the basic features.

R15d. Implementation, Snapshot, Testing
Flask Interface Snapshot Description:
/salary-bands shows three filters (Title, City, Term). After submitting, a table appears with Hourly Rate, Percentile, and Band for each matching posting.

Testing:

- Sample Data: Insert 3–5 postings with different hourly rates. Run the query and verify percentiles go from 0.00 to 1.00 and deciles 1 to 10.
- Sparse Case: If only 1–2 postings match, the page shows: "Not enough data to compute percentiles."
- Production: Run the same query on a real filter (e.g., SWE Intern, Toronto, Summer 2026) and confirm values look reasonable.