CPS510 Project – Job Bank

Nithursan Jeyabalasingam

Table of Contents

Project Description …………………………………………………………………….………… 3

ER Model ……………………………………………………………………………………...… 8

ER/EER Diagram …………………………………………………………………........................... 9

SQL – Simple Queries…………………………………………………………………………….. 10

SQL Queries using Relational Algebra ………………………………………………………… 17

SQL Queries by Unix Shell with Relational Algebra ……………………………………….… 19

Normalizing Tables to 3NF …………………………………………………………………..… 27

Normalizing Tables to BCNF ……………………………………………………………...… 32

Job Bank GUI Setup Instructions ……………………………………………………………… 34

Job Bank GUI Demo & Query Results from GUI ……………………………………...… 41

Lab 2 Assignment 1 – Application Description

Our topic for this assignment will be the **Online Job Bank System**. To build a successful application, a general description of the program as well as some requirements/functionalities must be planned beforehand to ensure that they are implemented properly during the development stage.

Description

This application should provide the general public with a platform that allows users to search for jobs that cater to their individual skills while prioritizing certain jobs based on their preferences. Some of these criteria may include:

- Profession (ex. engineer, teacher, mechanic, accountant, etc.)
- Salary (range sliders or boxes should be present, hourly wage and annual salaries implemented)
- Location (within the city, province/state, or global)
- Date job was posted
- Duration of job (permanent, part-time, seasonal, etc.)
- Work arrangement (onsite, remote, or hybrid)
- Required experience level (# of years the user should have in this field to qualify)
- Educational requirements (high school diploma, college/university degree, etc.)
- Application deadline date

In addition to these parameters, the program would ideally contain a description of the job written by the employer so that the user can decide if the job is right for them.

System Functions

A job bank application must be intuitive to use for both the end-user (people applying for the job) and for employers who wish to add job openings to the database. To accomplish this, a DBMS will be used that implements the following features:

- A view for employers that includes an intuitive menu to add jobs that contain all information listed in the "Description" section of this report.
- A view for users to filter through job openings that employers have posted.
- Employers should be able to edit their own job postings when needed.
- Once a user applies for a job, it should be removed from the list of available jobs for their view only so that other applicants using the program can still apply.
- Job applicants should have a separate view to see the status of any jobs they have applied to.

Requirements & Expectations

There are many requirements/expectations for a job bank DBMS to ensure that it both works reliably and as intended by all users. Some such requirements include:

- **Data Integrity**

  All data residing in the database should be valid and realistic. For example, the date the job was posted cannot be in the future.

- **Security Enforcement**

  Users of this application should only be able to edit data that they have permission to edit. For example, applicants looking for jobs should not be able to edit the details of job postings. Employers should also only be able to edit their own postings and not those of other employers.

- **Support for Standards**

  Data in the database should follow a predetermined standard to ensure all data is displayed properly. For example, dates should be in the SQL “DATE” format (our project will use the TO_DATE function with the format of ‘MMDDYY’), and descriptions of jobs should be strings, which in SQL, must be of data type “VARCHAR2”.

As the program is developed, more requirements may be found and implemented. All data present in this application should follow the SQL relational model, where all data is viewed in tables.

Entities & Relationships

Before constructing the database, the main entities and relationships must be established. Listed below are some of the entities that have been identified along with their respective primary and foreign keys are:

- USER (primary key: userID; no foreign keys)
- EMPLOYER (primary key: employerID; foreign key: userID)
- ADMINISTRATOR (primary key: administratorID; foreign key: userID)
- JOB (primary key: jobID; foreign key: employerID)
- APPLICATION (primary key: applicationID; foreign keys: applicantID, resumeID)
- APPLICANT (primary key: applicantID; foreign key: userID)
- RESUME (primary key: resumeID; foreign key: applicantID)
- APPLICANT_QUALIFICATIONS (primary key: applicant_qualificationsID; foreign key: resumeID)
- JOB_QUALIFICATIONS (primary key: job_qualificationsID; foreign key: jobID)

- LOCATION (primary key: locationID; foreign key: jobID)
- HR_EMPLOYEE (primary key: employeeID; no foreign keys)

Along with these entities, the following relationships have been recognized:

- **JOB_BANK & USER**

This relationship is a one-to-many relationship, meaning that multiple users can be part of the job bank. Both entities are connected via a "has" relationship, where "USER" participation is total but "JOB_BANK" participation is partial, since all users are part of the job bank but the job bank may have no users (such as when it is first made).

- **USER & EMPLOYER**

This relationship is a one-to-one relationship, meaning that only one user can be mapped to a single employer. Both entities are connected via an "is_an" relationship, where "EMPLOYER" participation is total but "USER" participation is partial, since all employers are users but not all users are employers.

- **USER & ADMINISTRATOR**

This is again a one-to-one relationship, meaning that only one user can be mapped to a single administrator. An administrator will be able to make changes to other entities and their respective attributes, such as employers, jobs, applications (for jobs), and applicants (users applying for jobs), as described below. Both entities are connected via an "is_an" relationship, where "ADMINISTRATOR" participation is total but "USER" participation is partial, since all administrators are users but not all users are administrators.

- **ADMINISTRATOR & EMPLOYER, ADMINISTRATOR & APPLICATION, ADMINISTRATOR & APPLICANT**

These three relationships are all many-to-many relationships since multiple administrators should be able to edit multiple employers, applications, and applicant data tuples at any time. All of these relationships are partial on both sides of the relationship.

- **USER & APPLICANT**

This is also a one-to-one relationship, as only one user can be mapped to a single applicant. Both entities are connected via an "is_an" relationship, where "APPLICANT" participation is total but "USER" participation is partial, since all applicants are users but not all users are applicants.

- **EMPLOYER & JOB**

  This relationship is a one-to-many relationship, as a single employer can post multiple job openings for different positions within a company. Both entities are connected via a “creates_a” relationship, where “EMPLOYER” participation is total but “JOB” participation is partial, since not all employers may have posted a job listing (such as when they first sign up), but all job listings are associated with an employer.

- **JOB & JOB_QUALIFICATIONS**

  This is also a one-to-many relationship, as a single job can have multiple qualifications that are required to be met by applicants. All of these relationships are total on both sides of the relationship since all jobs must have qualifications and all qualifications must be associated with a job.

- **JOB & LOCATION**

  This is a one-to-many relationship since a job can have multiple locations. Both entities are connected via a “has_a” relationship, where “LOCATION” participation is total but “JOB” participation is partial, since not all jobs have a location (such as when they are remote jobs), but all locations are associated with a job posting.

- **APPLICANT & RESUME**

  This is a one-to-one relationship, as an applicant can only have one resume. Both entities are connected via a “has_a” relationship, where “RESUME” participation is total but “APPLICANT” participation is partial, since all resumes are associated with an applicant but not all applicants have a resume.

- **RESUME & APPLICANT_QUALIFICATIONS**

  This is a one-to-many relationship since a resume may list multiple qualifications of the applicant. Both entities are connected via a “contains” relationship, where “APPLICANT_QUALIFICATIONS” participation is total but “RESUME” participation is partial, since there may be multiple qualifications listed in a resume, but not all resumes have qualifications (such as when the applicant has none).

- **APPLICANT & APPLICATION**

  This is a one-to-one relationship since an applicant can only send one application for a job posting. Both entities are connected via a “sends_an” relationship, where “APPLICATION” participation is total but “APPLICANT” participation is partial since every application for a job is associated with an applicant but an applicant may have no applications, such as when they first join the job bank.

- **RESUME & APPLICATION**

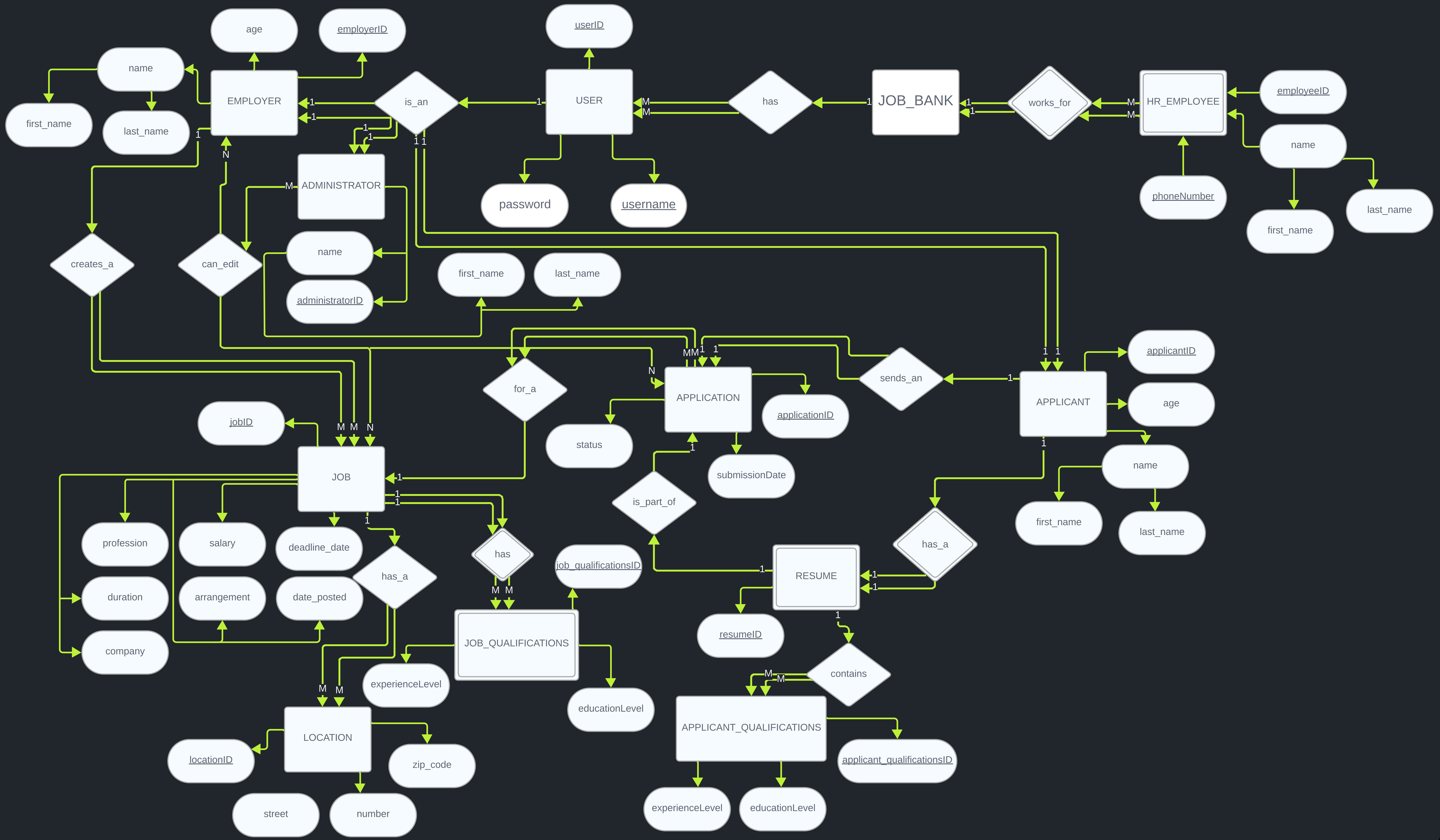
  This is a one-to-one relationship since an application for a job cannot contain multiple resumes. This is a partial participation on both sides of the relationship since not all resumes are part of an application and not all applications contain a resume (such as if the applicant chooses not to include one when applying).

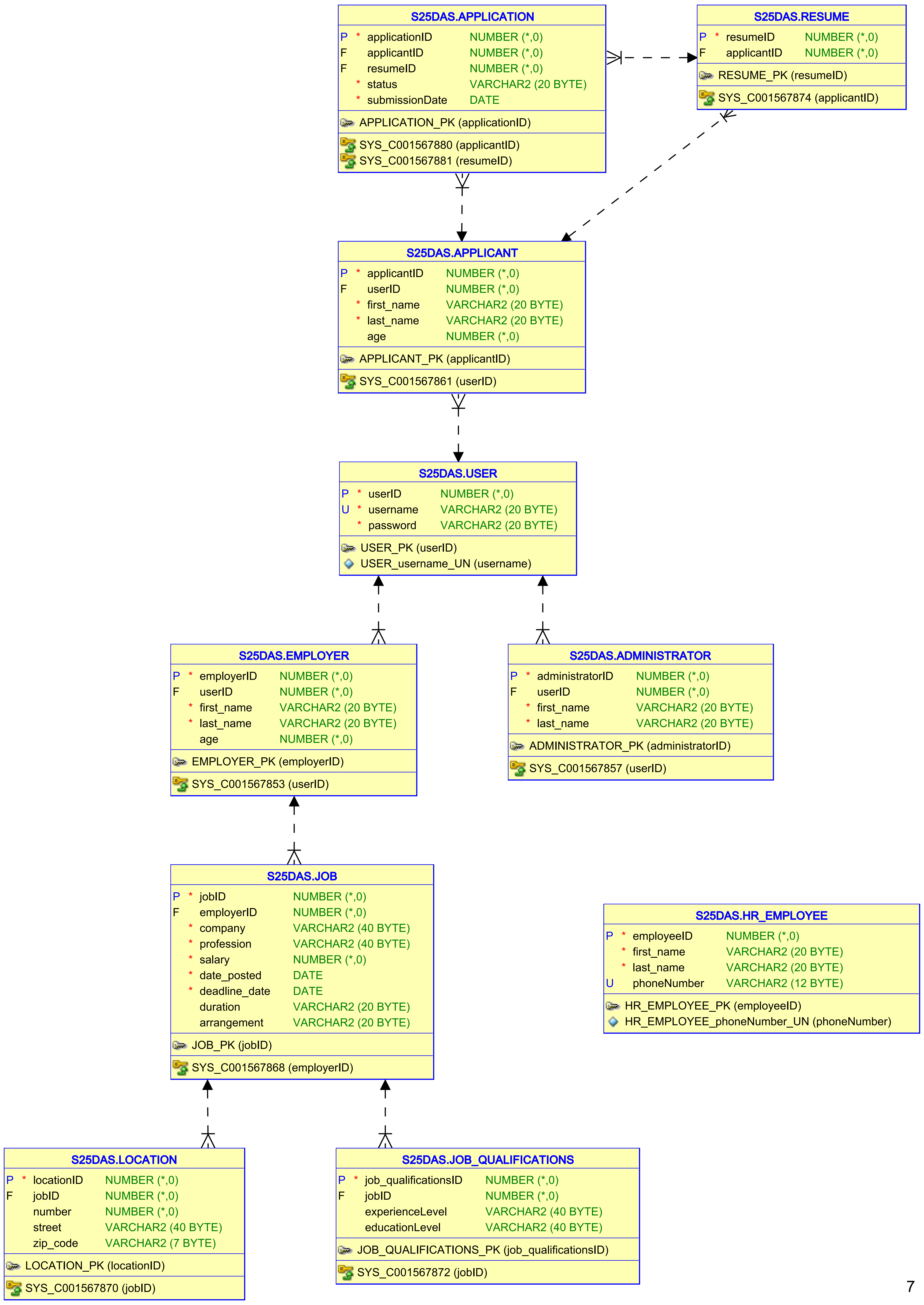
- **JOB & APPLICATION**

  This is a one-to-many relationship since a single job can have multiple applications from different applicants. Both entities are connected via a “for_a” relationship, where “APPLICATION” participation is total but “JOB” participation is partial since every application sent is for a job, but not all jobs have an application (such as in the case that no one applies for a posted job).

- **JOB_BANK & HR_EMPLOYEE**

  This is a one-to-many relationship since there is only one job bank, of which there can be multiple HR employees hired. Both entities are connected by a relationship of full participation on both sides since a job bank must have HR employees and employees must be part of the job bank.


age
employerID
userID
name
EMPLOYER
is_an
USER
has
JOB_BANK
works_for
HR_EMPLOYEE
employeeID
first_name
last_name
ADMINISTRATOR
password
username
name
phoneNumber
first_name
last_name
creates_a
can_edit
name
administratorID
first_name
last_name
for_a
APPLICATION
sends_an
APPLICANT
applicantID
age
applicationID
jobID
JOB
status
submissionDate
name
first_name
last_name
is_part_of
has_a
profession
salary
deadline_date
has
has_a
duration
arrangement
date_posted
job_qualificationsID
RESUME
company
JOB_QUALIFICATIONS
resumeID
contains
experienceLevel
educationLevel
LOCATION
APPLICANT_QUALIFICATIONS
locationID
zip_code
applicant_qualificationsID
street
number
experienceLevel
educationLevel



S25DAS.APPLICATION
P * applicationID NUMBER (*,0)
F applicantID NUMBER (*,0)
F resumeID NUMBER (*,0)
* status VARCHAR2 (20 BYTE)
* submissionDate DATE
APPLICATION_PK (applicationID)
SYS_C001567880 (applicantID)
SYS_C001567881 (resumeID)
S25DAS.RESUME
P * resumeID NUMBER (*,0)
F applicantID NUMBER (*,0)
RESUME_PK (resumeID)
SYS_C001567874 (applicantID)
S25DAS.APPLICANT
P * applicantID NUMBER (*,0)
F userID NUMBER (*,0)
* first_name VARCHAR2 (20 BYTE)
* last_name VARCHAR2 (20 BYTE)
age NUMBER (*,0)
APPLICANT_PK (applicantID)
SYS_C001567861 (userID)
S25DAS.USER
P * userID NUMBER (*,0)
U * username VARCHAR2 (20 BYTE)
* password VARCHAR2 (20 BYTE)
USER_PK (userID)
USER_username_UN (username)
S25DAS.EMPLOYER
P * employerID NUMBER (*,0)
F userID NUMBER (*,0)
* first_name VARCHAR2 (20 BYTE)
* last_name VARCHAR2 (20 BYTE)
age NUMBER (*,0)
EMPLOYER_PK (employerID)
SYS_C001567853 (userID)
S25DAS.ADMINISTRATOR
P * administratorID NUMBER (*,0)
F userID NUMBER (*,0)
* first_name VARCHAR2 (20 BYTE)
* last_name VARCHAR2 (20 BYTE)
ADMINISTRATOR_PK (administratorID)
SYS_C001567857 (userID)
S25DAS.JOB
P * jobID NUMBER (*,0)
F employerID NUMBER (*,0)
* company VARCHAR2 (40 BYTE)
* profession VARCHAR2 (40 BYTE)
* salary NUMBER (*,0)
* date_posted DATE
* deadline_date DATE
duration VARCHAR2 (20 BYTE)
arrangement VARCHAR2 (20 BYTE)
JOB_PK (jobID)
SYS_C001567868 (employerID)
S25DAS.HR_EMPLOYEE
P * employeeID NUMBER (*,0)
* first_name VARCHAR2 (20 BYTE)
* last_name VARCHAR2 (20 BYTE)
U phoneNumber VARCHAR2 (12 BYTE)
HR_EMPLOYEE_PK (employeeID)
HR_EMPLOYEE_phoneNumber_UN (phoneNumber)
S25DAS.LOCATION
P * locationID NUMBER (*,0)
F jobID NUMBER (*,0)
number NUMBER (*,0)
street VARCHAR2 (40 BYTE)
zip_code VARCHAR2 (7 BYTE)
LOCATION_PK (locationID)
SYS_C001567870 (jobID)
S25DAS.JOB_QUALIFICATIONS
P * job_qualificationsID NUMBER (*,0)
F jobID NUMBER (*,0)
experienceLevel VARCHAR2 (40 BYTE)
educationLevel VARCHAR2 (40 BYTE)
JOB_QUALIFICATIONS_PK (job_qualificationsID)
SYS_C001567872 (jobID)

Assignment 4 Part 1 - Section 7, Topic 4 Job Bank

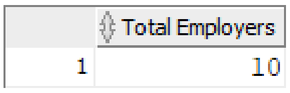
In this assignment, some example queries for our SQL database will be demonstrated, along with the source code for each query, and finally an explanation as to how the query functions.

Query 1: Display Total Number of Employers

- **Source Code:**

```
SELECT COUNT(DISTINCT "employerID") AS "Total Employers" FROM "EMPLOYER";
```

- **Output:**



```
     Total Employers
1                 10
```

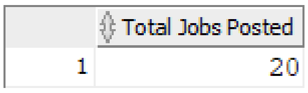
- **Explanation:**
This query counts the total number of employers present in the database. In the source code for this query, FROM “EMPLOYER” means that the data will be pulled from the “EMPLOYER” table. The code AS “Total Employers” means that the column that will display the data being pulled will be titled “Total Employers”, as shown in the output screenshot. Finally the COUNT(DISTINCT “employerID”) means that only distinct (unique) employer IDs will be counted in the data, and the SELECT statement means that tuples that match this condition will be displayed in the query output.

Query 2: Display Total Number of Jobs Posted

- **Source Code:**

```
SELECT COUNT(DISTINCT "jobID") AS "Total Jobs Posted" FROM "JOB";
```

- **Output:**



```
     Total Jobs Posted
1                   20
```

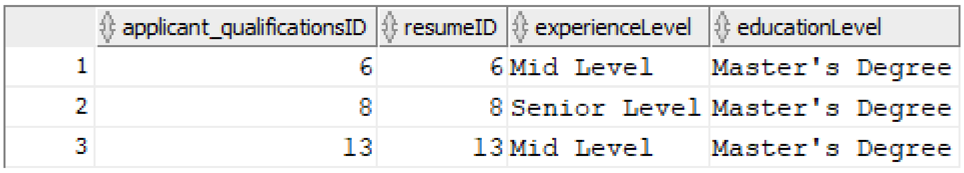
- **Explanation:**
This query counts the total number of jobs present in the database. In the source code for this query, FROM “JOB” means that the data will be pulled from the “JOB” table. The code AS “Total Jobs Posted” means that the column that will display the data being pulled will be titled “Total Jobs Posted”, as shown in the output screenshot. Finally the COUNT(DISTINCT “jobID”) means that only distinct (unique) job IDs will be counted in the data, and the SELECT statement means that tuples that match this condition will be displayed in the query output.

Query 3: Display All Applicants that have a Master’s Degree

- **Source Code:**

```
SELECT *
FROM "APPLICANT_QUALIFICATIONS"
WHERE "educationLevel" = 'Master''s Degree';
```

- **Output:**



	applicant_qualificationsID	resumeID	experienceLevel	educationLevel
1	6	6	Mid Level	Master's Degree
2	8	8	Senior Level	Master's Degree
3	13	13	Mid Level	Master's Degree

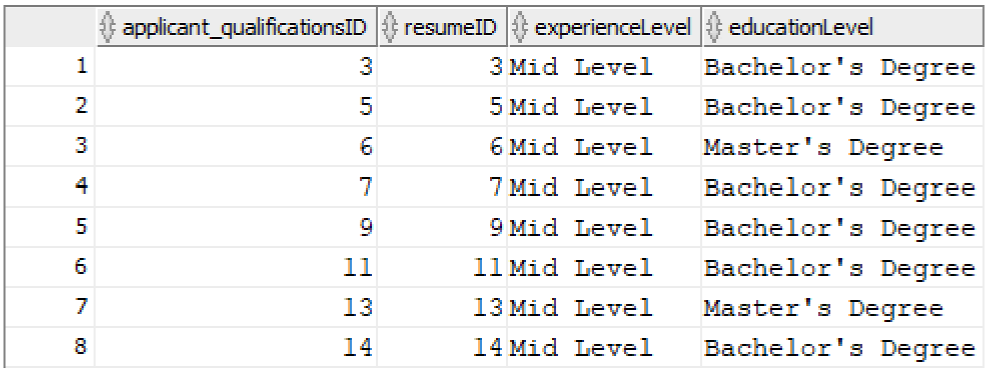
- **Explanation:**
This query displays all applicants for jobs that have a master’s degree listed in their qualifications. In the source code, FROM “APPLICANT_QUALIFICATIONS” means that the data for this query will be pulled from the “APPLICANT_QUALIFICATIONS” table in the database. The code WHERE “educationLevel” = ‘Master’’s Degree’ means that in order for a tuple to be listed in the query results, the education level for the applicant must be exactly equal to a master’s degree. It is also worth noting that in this query, an escape character is used before the apostrophe to prevent a syntax error in the string value “Master’s Degree”. Finally, the code SELECT * means all tuples that meet these criteria will be listed in the query output.

Query 4: Display Applicants that have a Mid Level Experience

- **Source Code:**

```
SELECT *
FROM "APPLICANT_QUALIFICATIONS"
WHERE "experienceLevel" = 'Mid Level';
```

- **Output:**



	applicant_qualificationsID	resumeID	experienceLevel	educationLevel
1	3	3	Mid Level	Bachelor's Degree
2	5	5	Mid Level	Bachelor's Degree
3	6	6	Mid Level	Master's Degree
4	7	7	Mid Level	Bachelor's Degree
5	9	9	Mid Level	Bachelor's Degree
6	11	11	Mid Level	Bachelor's Degree
7	13	13	Mid Level	Master's Degree
8	14	14	Mid Level	Bachelor's Degree

- **Explanation:**
This query displays all applicants that have an experience level of “Mid Level”. Firstly, FROM “APPLICANT_QUALIFICATIONS” means that the data will be taken from the “APPLICANT_QUALIFICATIONS” table. The code WHERE “experienceLevel” = ‘Mid Level’ means that all applicants whose attribute for

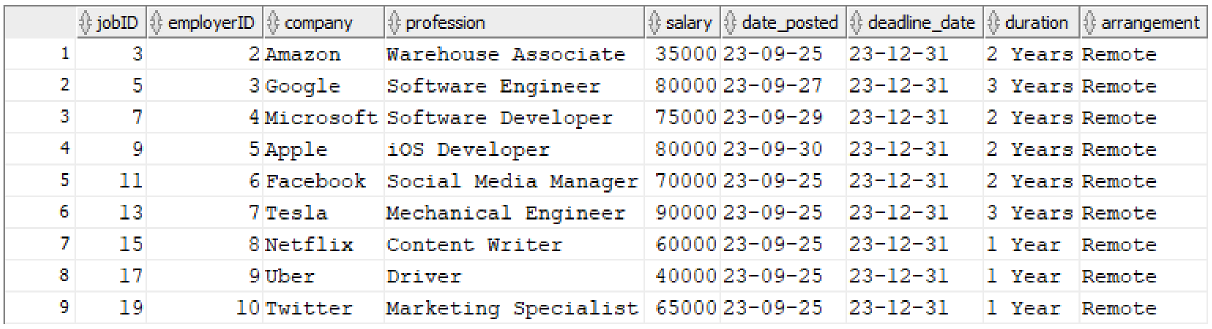
experience level equal to “Mid Level” will be listed in the query result. Finally, SELECT * means that all applicants that fit this description will be listed.

Query 5: Display All Remote Jobs

- **Source Code:**

```
SELECT *
FROM "JOB"
WHERE "arrangement" = 'Remote';
```

- **Output:**



	jobID	employerID	company	profession	salary	date_posted	deadline_date	duration	arrangement
1	3	2	Amazon	Warehouse Associate	35000	23-09-25	23-12-31	2 Years	Remote
2	5	3	Google	Software Engineer	80000	23-09-27	23-12-31	3 Years	Remote
3	7	4	Microsoft	Software Developer	75000	23-09-29	23-12-31	2 Years	Remote
4	9	5	Apple	iOS Developer	80000	23-09-30	23-12-31	2 Years	Remote
5	11	6	Facebook	Social Media Manager	70000	23-09-25	23-12-31	2 Years	Remote
6	13	7	Tesla	Mechanical Engineer	90000	23-09-25	23-12-31	3 Years	Remote
7	15	8	Netflix	Content Writer	60000	23-09-25	23-12-31	1 Year	Remote
8	17	9	Uber	Driver	40000	23-09-25	23-12-31	1 Year	Remote
9	19	10	Twitter	Marketing Specialist	65000	23-09-25	23-12-31	1 Year	Remote

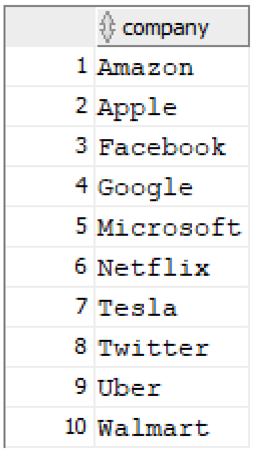
- **Explanation:**
This query displays all jobs in the database that are remote. This query will pull data from the “JOB” table with the code FROM “JOB”, and the code WHERE “arrangement” = ‘Remote’ means that only jobs that have the arrangement of remote will be listed. The SELECT * statement will only select jobs matching this criteria to be listed in the query output.

Query 6: Display All Companies in Alphabetical Order

- **Source Code:**

```
SELECT DISTINCT "company"
FROM "JOB"
ORDER BY "company" ASC;
```

- **Output:**



	company
1	Amazon
2	Apple
3	Facebook
4	Google
5	Microsoft
6	Netflix
7	Tesla
8	Twitter
9	Uber
10	Walmart

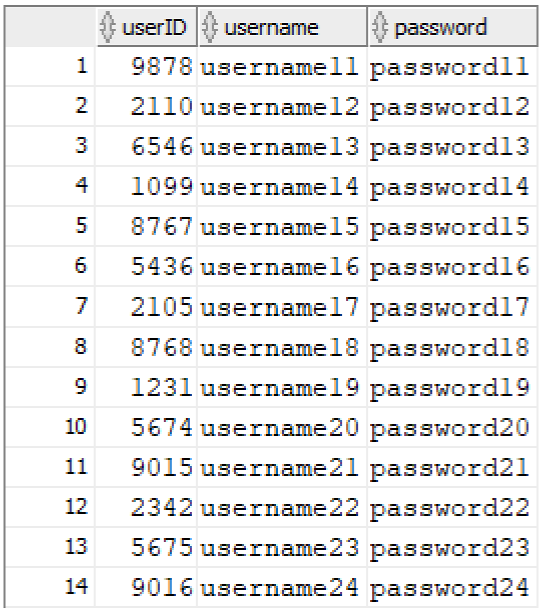
- **Explanation:**
  This query displays all companies in the database that have a job listing in alphabetical order. FROM “JOB” means data will be pulled from the “JOB” table. The code ORDER BY “company” ASC means that the job table contains an attribute called “company”, and the query output will order the companies based on this attribute in ascending (alphabetical) order based off this attribute. SELECT DISTINCT “company” will ensure that no duplicate companies are listed in the output, since a company can have multiple job openings in the job bank which must be filtered for this query.

Query 7: Display all Applicants’ userID, username, and password

- **Source Code:**

```
SELECT U.*
FROM "USER" U
INNER JOIN "APPLICANT" A ON U."userID" = A."userID";
```

- **Output:**

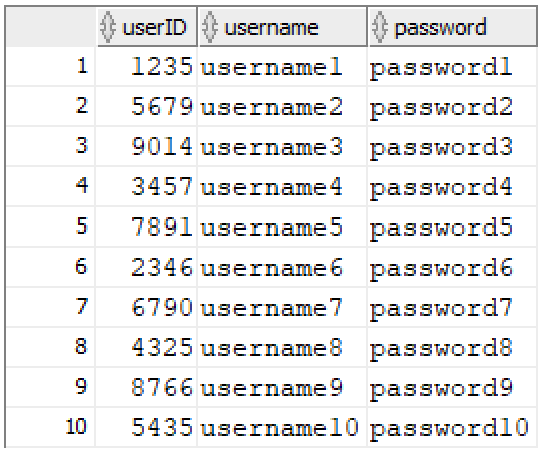


	userID	username	password
1	9878	username11	password11
2	2110	username12	password12
3	6546	username13	password13
4	1099	username14	password14
5	8767	username15	password15
6	5436	username16	password16
7	2105	username17	password17
8	8768	username18	password18
9	1231	username19	password19
10	5674	username20	password20
11	9015	username21	password21
12	2342	username22	password22
13	5675	username23	password23
14	9016	username24	password24

- **Explanation:**
  This query displays all applicant’s userID, username, and password in the database. The code FROM “USER” U means that data will be pulled from the “USER” table and will be assigned to the temporary variable U. The code INNER JOIN “APPLICANT” A ON U.”userID” = A.”userID” means that in order to be listed in the query output, the userID in the “APPLICANT” table must match the userID in the “USER” table. If this criteria is met, the two tables will be joined to display the username and password. The code SELECT U.* means this check will be performed for all tuples in the “USER” table.

Query 8: Display all Employers’ userID, username, and password

- **Source Code:**
```
SELECT U.*
FROM "USER" U
INNER JOIN "EMPLOYER" E ON U."userID" = E."userID";
```
- **Output:**

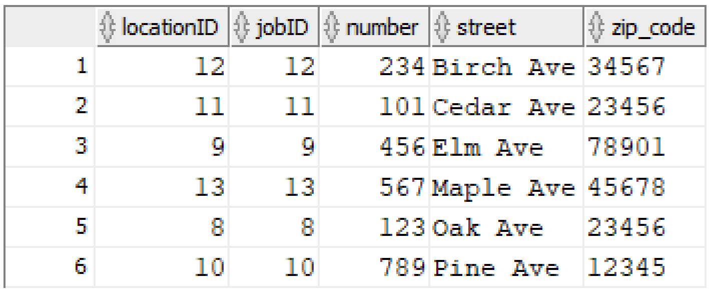


	userID	username	password
1	1235	username1	password1
2	5679	username2	password2
3	9014	username3	password3
4	3457	username4	password4
5	7891	username5	password5
6	2346	username6	password6
7	6790	username7	password7
8	4325	username8	password8
9	8766	username9	password9
10	5435	username10	password10

- **Explanation:**
This query is similar to query 6, except this query compares users against the “EMPLOYER” table instead of the “APPLICANT” table.

Query 9: Display All Jobs Located on an Avenue in Alphabetical Order

- **Source Code:**
```
SELECT *
FROM "LOCATION"
WHERE "street" LIKE '%Ave%'
ORDER BY "street" ASC;
```
- **Output:**



	locationID	jobID	number	street	zip_code
1	12	12	234	Birch Ave	34567
2	11	11	101	Cedar Ave	23456
3	9	9	456	Elm Ave	78901
4	13	13	567	Maple Ave	45678
5	8	8	123	Oak Ave	23456
6	10	10	789	Pine Ave	12345

- **Explanation:**
This query displays all locations that are listed on an avenue. To perform this query, firstly, FROM “LOCATION” indicated that data will be pulled from the “LOCATION” table in the database. The code WHERE “street” LIKE ‘%Ave%’ means that it will check the attribute “street” in the “LOCATION” table and search for “Ave” anywhere in the string. The “%” is a wildcard symbol that indicates that any

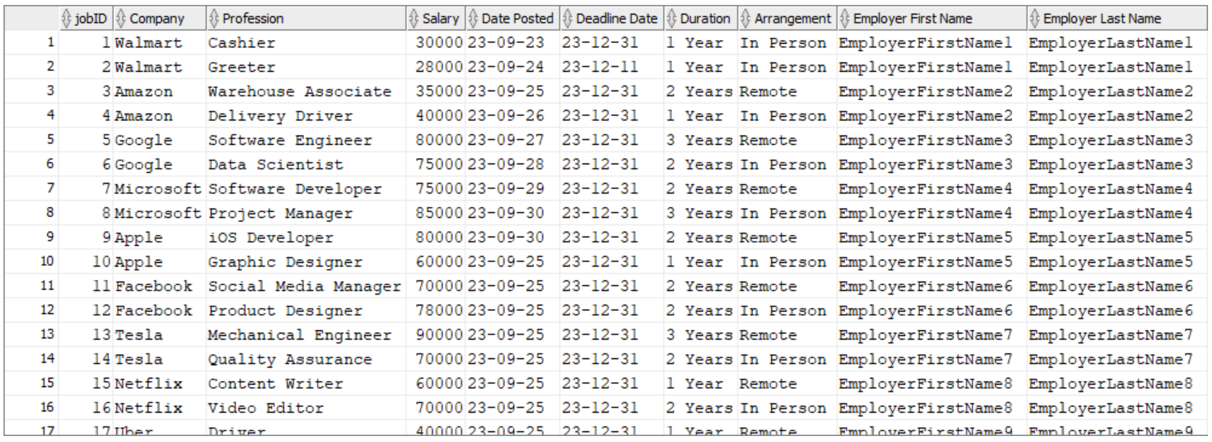
characters can be before or after the “Ave” string in order to match the criteria. This will ensure that streets of different names can still be listed. Finally, the code ORDER BY “street” ASC means that the streets will be listed in the query output in alphabetical order.

Query 10: Display all Jobs alongside Employer Information

- **Source Code:**

```
SELECT
    J."jobID",
    J."company" AS "Company",
    J."profession" AS "Profession",
    J."salary" AS "Salary",
    J."date_posted" AS "Date Posted",
    J."deadline_date" AS "Deadline Date",
    J."duration" AS "Duration",
    J."arrangement" AS "Arrangement",
    E."first_name" AS "Employer First Name",
    E."last_name" AS "Employer Last Name"
FROM "JOB" J
INNER JOIN "EMPLOYER" E ON J."employerID" = E."employerID";
```

- **Output (cut off due to too many entries):**



	jobID	Company	Profession	Salary	Date Posted	Deadline Date	Duration	Arrangement	Employer First Name	Employer Last Name
1	1	Walmart	Cashier	30000	23-09-23	23-12-31	1 Year	In Person	EmployerFirstName1	EmployerLastName1
2	2	Walmart	Greeter	28000	23-09-24	23-12-11	1 Year	In Person	EmployerFirstName1	EmployerLastName1
3	3	Amazon	Warehouse Associate	35000	23-09-25	23-12-31	2 Years	Remote	EmployerFirstName2	EmployerLastName2
4	4	Amazon	Delivery Driver	40000	23-09-26	23-12-31	1 Year	In Person	EmployerFirstName2	EmployerLastName2
5	5	Google	Software Engineer	80000	23-09-27	23-12-31	3 Years	Remote	EmployerFirstName3	EmployerLastName3
6	6	Google	Data Scientist	75000	23-09-28	23-12-31	2 Years	In Person	EmployerFirstName3	EmployerLastName3
7	7	Microsoft	Software Developer	75000	23-09-29	23-12-31	2 Years	Remote	EmployerFirstName4	EmployerLastName4
8	8	Microsoft	Project Manager	85000	23-09-30	23-12-31	3 Years	In Person	EmployerFirstName4	EmployerLastName4
9	9	Apple	iOS Developer	80000	23-09-30	23-12-31	2 Years	Remote	EmployerFirstName5	EmployerLastName5
10	10	Apple	Graphic Designer	60000	23-09-25	23-12-31	1 Year	In Person	EmployerFirstName5	EmployerLastName5
11	11	Facebook	Social Media Manager	70000	23-09-25	23-12-31	2 Years	Remote	EmployerFirstName6	EmployerLastName6
12	12	Facebook	Product Designer	78000	23-09-25	23-12-31	2 Years	In Person	EmployerFirstName6	EmployerLastName6
13	13	Tesla	Mechanical Engineer	90000	23-09-25	23-12-31	3 Years	Remote	EmployerFirstName7	EmployerLastName7
14	14	Tesla	Quality Assurance	70000	23-09-25	23-12-31	2 Years	In Person	EmployerFirstName7	EmployerLastName7
15	15	Netflix	Content Writer	60000	23-09-25	23-12-31	1 Year	Remote	EmployerFirstName8	EmployerLastName8
16	16	Netflix	Video Editor	70000	23-09-25	23-12-31	2 Years	In Person	EmployerFirstName8	EmployerLastName8
17	17	Uber	Driver	40000	23-09-25	23-12-31	1 Year	Remote	EmployerFirstName9	EmployerLastName9

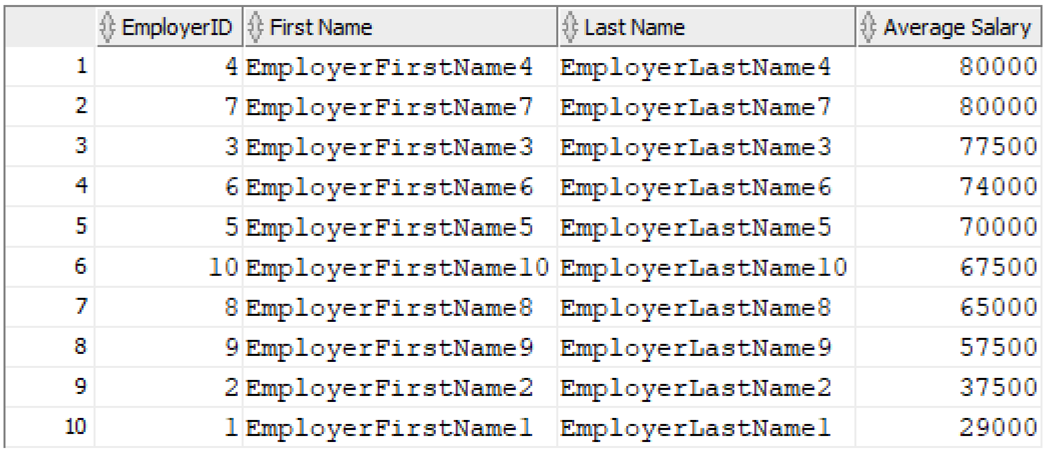
- **Explanation:**
This query displays all jobs along with the corresponding first and last name of the employer for that job. The SELECT statement sets the column titles for each attribute in the “JOB” table using the FROM “JOB” J statement. The code INNER JOIN “EMPLOYER” E ON J.”employerID” = E.”employerID” means that if the job employer ID matches an employer’s employer ID, that employer created that job and therefore the two tuples should be joined.

Query 11: Display Average Salary for Jobs Posted by Each Employer in Descending Order

- **Source Code:**

```
SELECT
    E."employerID" AS "EmployerID",
    E."first_name" AS "First Name",
    E."last_name" AS "Last Name",
    AVG(J."salary") AS "Average Salary"
FROM "EMPLOYER" E
INNER JOIN "JOB" J ON E."employerID" = J."employerID"
GROUP BY E."employerID", E."first_name", E."last_name"
HAVING COUNT(J."jobID") > 0
ORDER BY "Average Salary" DESC;
```

- **Output:**



	EmployerID	First Name	Last Name	Average Salary
1	4	EmployerFirstName4	EmployerLastName4	80000
2	7	EmployerFirstName7	EmployerLastName7	80000
3	3	EmployerFirstName3	EmployerLastName3	77500
4	6	EmployerFirstName6	EmployerLastName6	74000
5	5	EmployerFirstName5	EmployerLastName5	70000
6	10	EmployerFirstName10	EmployerLastName10	67500
7	8	EmployerFirstName8	EmployerLastName8	65000
8	9	EmployerFirstName9	EmployerLastName9	57500
9	2	EmployerFirstName2	EmployerLastName2	37500
10	1	EmployerFirstName1	EmployerLastName1	29000

- **Explanation:**
  This query displays the average salary for jobs posted by each employer sorted in descending order. Firstly, the SELECT statement creates the column titles. The AVG statement will calculate the average for each job. The statement FROM "EMPLOYER" E means that data will be pulled from the employer table. The code INNER JOIN "JOB" J ON E."employerID" = J."employerID" means that if the employer ID in the employer table matches the employer ID in the job table, that row will be used. The GROUP BY statement will group together employer IDs, first names, and last names to ensure each employer only has one row in the query output. The HAVING COUNT statement will exclude any employers who have not posted a job yet. Finally, the ORDER BY statement will sort the salaries in descending order.

Total Employers:

- Relational Algebra:
  $\pi_{\text{"Total Employers"}}$("EMPLOYER")

Total Jobs Posted:

- Relational Algebra:
  $\pi_{\text{"Total Jobs Posted"}}$("JOB")

Jobs Requiring Master's Degrees:

- Relational Algebra:
  $\sigma_{\text{"educationLevel"='Master''s Degree'}}$("APPLICANT_QUALIFICATIONS")

Applicants with Mid-Level Experience:

- Relational Algebra:
  $\sigma_{\text{"experienceLevel"='Mid Level'}}$("APPLICANT_QUALIFICATIONS")

Remote Jobs:

- Relational Algebra:
  $\sigma_{\text{"arrangement"='Remote'}}$("JOB")

Alphabetical Order of Companies:

- Relational Algebra:
  $\pi_{\text{"company"}}$("JOB") $\bowtie_{\text{"company"} \neq \text{"company"}}$ $\pi_{\text{"company"}}$("JOB")

Applicants:

- Relational Algebra:
  $\pi_{\text{"USER".*}}$("USER" $\bowtie_{\text{"userID"="userID"}}$ $\pi_{\text{"APPLICANT".*}}$("APPLICANT"))

Employers:

- Relational Algebra:
  $\pi_{\text{"USER".*}}$("USER" $\bowtie_{\text{"userID"="userID"}}$ $\pi_{\text{"EMPLOYER".*}}$("EMPLOYER"))

Avenues:

- Relational Algebra:
- π(location_attributes)(σ (street LIKE '%Ave%') (LOCATION))

Job Table with Employer Names:

- Relational Algebra:

  π"jobID", "company", "profession", "salary", "date_posted", "deadline_date", "duration", "arrangement", "first_name", "last_name"("JOB"⋈"employerID"="employerID""EMPLOYER")

Average Salary of Employers in Descending Order:

- Relational Algebra:

  π"employerID", "first_name", "last_name", AVG("salary")(σ COUNT("jobID")>0 ("EMPLOYER"⋈"employerID"="employerID""JOB")⋈"employerID"="employerID"" EMPLOYER")

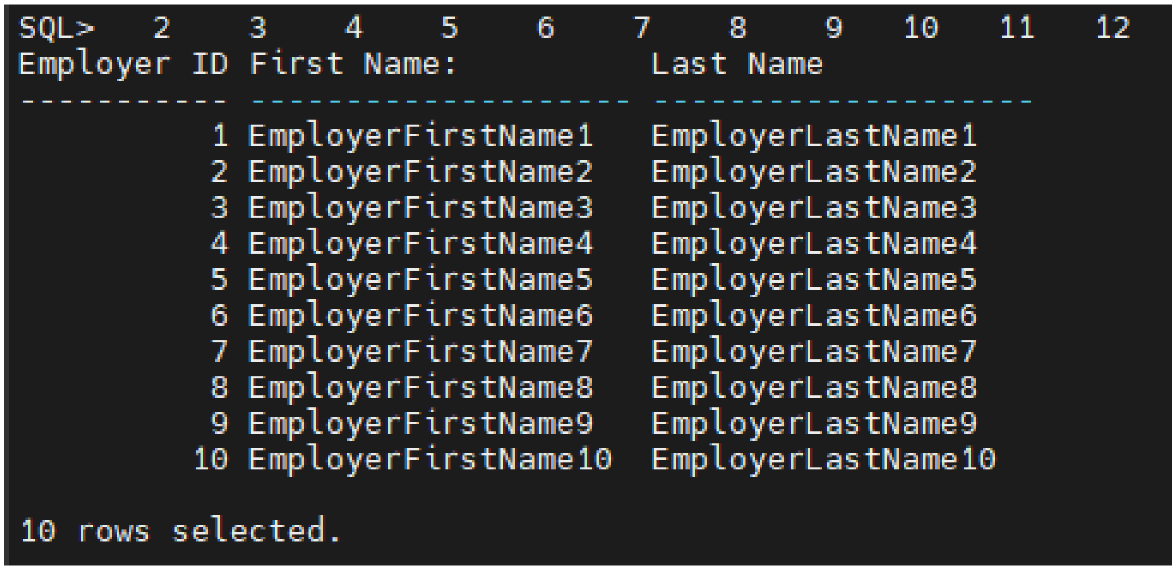
Assignment 5 - Section 7, Topic 4

Job Bank

In this assignment, some example queries for our SQL database will be demonstrated, along with the source code for each query, and finally an explanation as to how the query functions.

Query 1: Returns a list of employers who have posted jobs and, for at least one of those jobs, specified a location.

● **Source Code:**
```
SELECT e."employerID" AS "Employer ID", e."first_name" AS "First
Name:", e."last_name" AS "Last Name"
FROM "EMPLOYER" e
WHERE EXISTS (
    SELECT 1
    FROM "JOB" j
    WHERE j."employerID" = e."employerID"
    AND EXISTS (
        SELECT 1
        FROM "LOCATION" l
        WHERE l."jobID" = j."jobID"
    )
);
```
● **Output:**

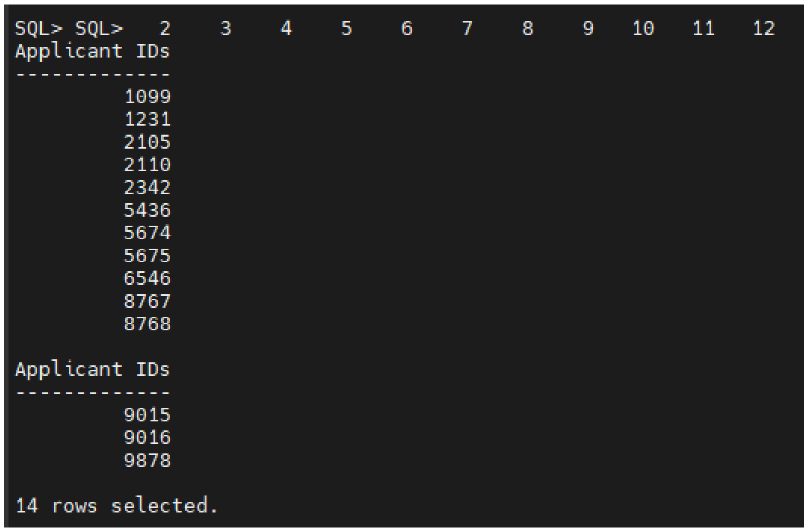


```
SQL>   2    3    4    5    6    7    8    9   10   11   12
Employer ID First Name:          Last Name
----------- -------------------- --------------------
          1 EmployerFirstName1   EmployerLastName1
          2 EmployerFirstName2   EmployerLastName2
          3 EmployerFirstName3   EmployerLastName3
          4 EmployerFirstName4   EmployerLastName4
          5 EmployerFirstName5   EmployerLastName5
          6 EmployerFirstName6   EmployerLastName6
          7 EmployerFirstName7   EmployerLastName7
          8 EmployerFirstName8   EmployerLastName8
          9 EmployerFirstName9   EmployerLastName9
         10 EmployerFirstName10  EmployerLastName10

10 rows selected.
```

● **Explanation:**
This query retrieves information about employers who have posted jobs with specified locations. It selects data from the "EMPLOYER" table and renames the columns as "Employer ID," "First Name," and "Last Name." The WHERE clause includes two nested EXISTS subqueries. The first subquery checks if there is at least one job posted by each employer in the "JOB" table. The second subquery, nested within the first, confirms the existence of at least one location specified for a job in the "LOCATION" table that corresponds to a job posted by the employer. In summary, the query identifies employers who have posted jobs and ensured that at least one of those jobs has a specified location. It returns their information, including the "Employer ID," "First Name," and "Last Name."

Query 2: Union of distinct usernames from "EMPLOYER" and "APPLICANT"

● **Source Code:**
```
-- Union of distinct usernames from "EMPLOYER" and "APPLICANT"
SELECT "userID" AS "Applicant IDs"
FROM "EMPLOYER" e
UNION
SELECT "userID"
FROM "APPLICANT" a
-- Minus the usernames that are common between "EMPLOYER" and "APPLICANT"
MINUS
SELECT "userID"
FROM "EMPLOYER" e
INTERSECT
SELECT "userID"
FROM "APPLICANT" a;
```
● **Output:**

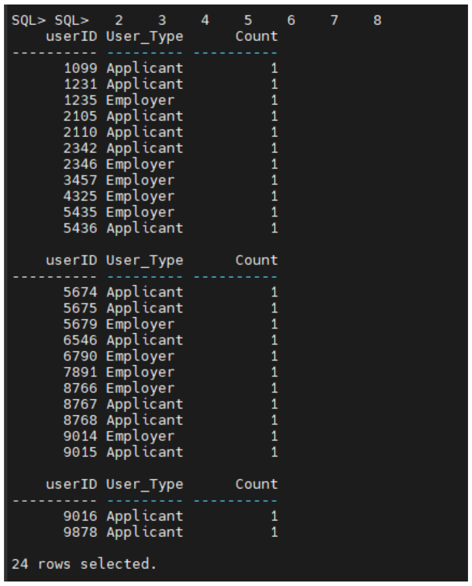

```
SQL> SQL>   2    3    4    5    6    7    8    9   10   11   12
Applicant IDs
-------------
         1099
         1231
         2105
         2110
         2342
         5436
         5674
         5675
         6546
         8767
         8768

Applicant IDs
-------------
         9015
         9016
         9878

14 rows selected.
```

● **Explanation:**
This query combines data from the "EMPLOYER" and "APPLICANT" tables to identify distinct user IDs, which can represent either employers or applicants. First, it employs a UNION operation to merge the distinct "userID" values from the "EMPLOYER" and "APPLICANT" tables into a single result set, assigning them the label "Applicant IDs." This ensures that all unique user IDs from both tables are included once in the output.

Next, the code uses the MINUS and INTERSECT set operators to refine the results. The MINUS operation subtracts the common user IDs between the "EMPLOYER" and "APPLICANT" tables, effectively eliminating users who are both employers and applicants. Conversely, the INTERSECT operation finds the user IDs that exist in both tables, representing users who are both employers and applicants.

Query 3: Categorize the number of employers and applicants for each unique username

● **Source Code:**
```
-- Count the number of employers for each unique username
SELECT "userID", 'Employer' AS "User_Type", COUNT(*) AS "Count"
FROM "EMPLOYER" e
GROUP BY "userID"
UNION
-- Count the number of applicants for each unique username
SELECT "userID", 'Applicant' AS "User_Type", COUNT(*) AS "Count"
FROM "APPLICANT" a
GROUP BY "userID";
```
● **Output:**

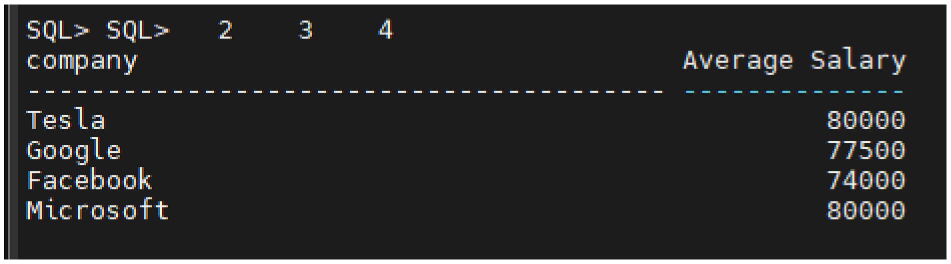


```
SQL> SQL>   2    3    4    5    6    7    8
    userID User_Type      Count
---------- --------- ----------
      1099 Applicant          1
      1231 Applicant          1
      1235 Employer           1
      2105 Applicant          1
      2110 Applicant          1
      2342 Applicant          1
      2346 Employer           1
      3457 Employer           1
      4325 Employer           1
      5435 Employer           1
      5436 Applicant          1

    userID User_Type      Count
---------- --------- ----------
      5674 Applicant          1
      5675 Applicant          1
      5679 Employer           1
      6546 Applicant          1
      6790 Employer           1
      7891 Employer           1
      8766 Employer           1
      8767 Applicant          1
      8768 Applicant          1
      9014 Employer           1
      9015 Applicant          1

    userID User_Type      Count
---------- --------- ----------
      9016 Applicant          1
      9878 Applicant          1

24 rows selected.
```

● **Explanation:**
This query is used to count and categorize the number of employers and applicants for each unique username. It achieves this through a series of SQL operations. First, it uses a SELECT statement to count the number of employers by selecting the "userID" from the "EMPLOYER" table and assigns a label "User_Type" with the value 'Employer' for each record. The COUNT(*) function is applied to calculate the count of employers for each unique username. This information is grouped using the GROUP BY clause, which ensures that each unique username becomes a distinct group.

Similarly, the code then proceeds to count the number of applicants by selecting the "userID" from the "APPLICANT" table, assigning the "User_Type" label as 'Applicant,' and counting the number of applicants for each unique username with COUNT(*). The results from both operations are combined using the UNION operator to form a single result set, providing a comprehensive count of employers and applicants for each unique username, along with their respective user types ('Employer' or 'Applicant'). This information can be helpful in analyzing the distribution of users based on their roles in the system.

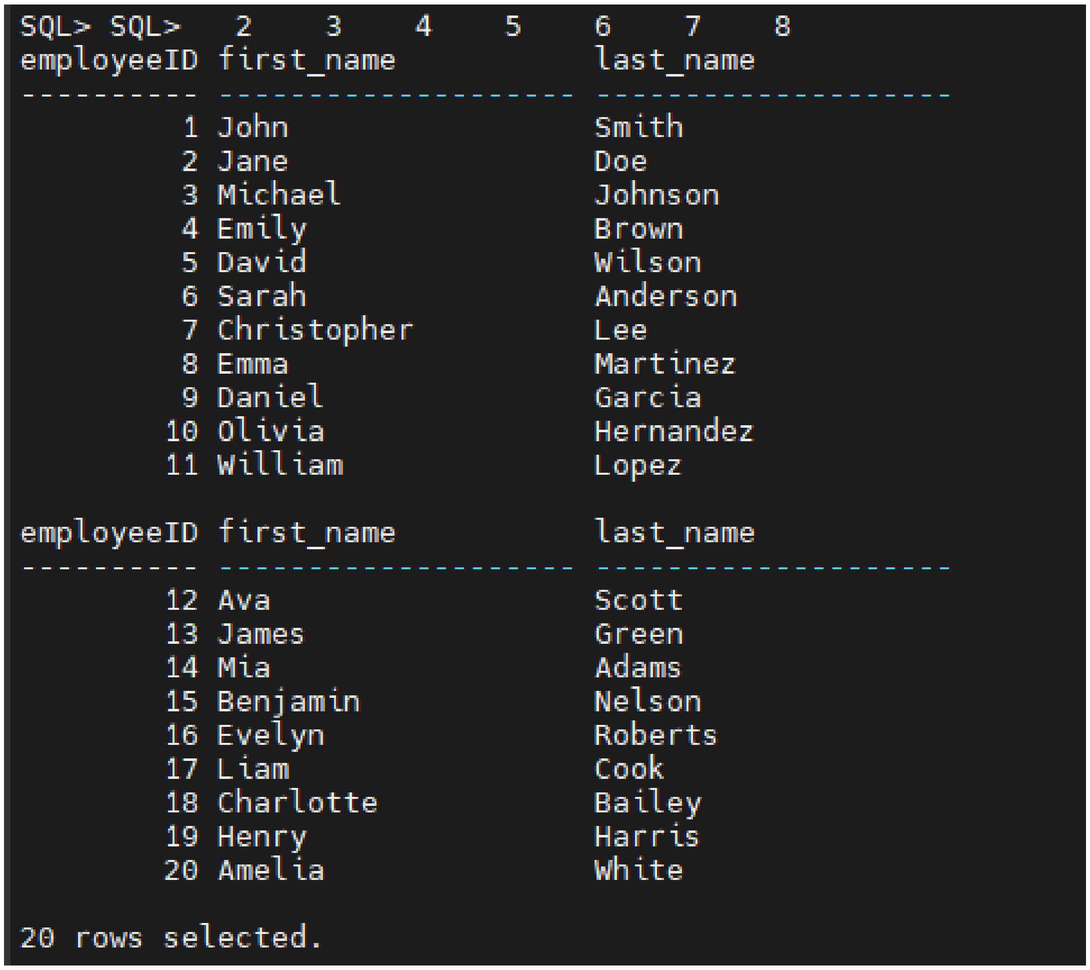
Query 4: Average of Salaries of Companies where the average > 70000

● **Source Code:**
```
SELECT "company", AVG("salary") AS "Average Salary"
FROM "JOB"
HAVING AVG("salary") > 70000
GROUP BY "company";
```
● **Output:**

```
SQL> SQL>   2    3    4
company                                  Average Salary
---------------------------------------- --------------
Tesla                                             80000
Google                                            77500
Facebook                                          74000
Microsoft                                         80000
```
● **Explanation:**
The provided SQL code is used to retrieve information about companies that post job listings with an average salary greater than $70,000. It calculates the average salary for each company by selecting the "company" and applying the AVG("salary") function to calculate the average salary for jobs offered by that company. The GROUP BY clause is used to group the results by the "company" column, ensuring that each company is treated as a distinct group.

The HAVING clause is then applied to filter the results. It allows only those groups (companies) where the average salary, as calculated by AVG("salary"), is greater than $70,000 to be included in the output. This is used to identify companies that offer jobs with higher-than-average salaries. The query produces a result set that displays the "company" and their corresponding "Average Salary," showing only companies that meet the specified salary threshold. This information can be valuable for companies and job seekers looking for positions with higher salary expectations.

Query 5: HR employees who have a phone number different from the default value.

● **Source Code:**
```
SELECT "employeeID", "first_name", "last_name"
FROM "HR_EMPLOYEE" e
WHERE EXISTS (
    SELECT 1
    FROM "HR_EMPLOYEE" sub
    WHERE e."employeeID" = sub."employeeID"
    AND sub."phoneNumber" <> '-1'
);
```
● **Output:**



```
SQL> SQL>   2    3    4    5    6    7    8
employeeID first_name           last_name
---------- -------------------- --------------------
         1 John                 Smith
         2 Jane                 Doe
         3 Michael              Johnson
         4 Emily                Brown
         5 David                Wilson
         6 Sarah                Anderson
         7 Christopher          Lee
         8 Emma                 Martinez
         9 Daniel               Garcia
        10 Olivia               Hernandez
        11 William              Lopez

employeeID first_name           last_name
---------- -------------------- --------------------
        12 Ava                  Scott
        13 James                Green
        14 Mia                  Adams
        15 Benjamin             Nelson
        16 Evelyn               Roberts
        17 Liam                 Cook
        18 Charlotte            Bailey
        19 Henry                Harris
        20 Amelia               White

20 rows selected.
```

● **Explanation:**
The provided SQL query is designed to retrieve data from the "HR_EMPLOYEE" table, specifically the "employeeID," "first_name," and "last_name" of HR employees who have a non-default phone number. It accomplishes this by utilizing the EXISTS subquery. The main query selects the mentioned columns from the "HR_EMPLOYEE" table, labeled as "e." It then utilizes the EXISTS clause to check for the existence of at least one record in the same table, referred to as "sub," where the "employeeID" matches that of the main query and where the "phoneNumber" is different from the default value of '-1.' This effectively identifies HR employees with non-default phone numbers. The query's output provides the "employeeID," "first_name," and "last_name" of these HR employees who meet this condition. This information can be valuable for HR management or reporting purposes, such as identifying employees with updated contact information.

Relational Algebra for each Query:

Query 1: Returns a list of employers who have posted jobs and, for at least one of those jobs, specified a location:
π"employerID","first_name","last_name"(σEXISTS((σ"jobID"="jobID"("LOCATION")⋈"employerID"="employerID"σEXISTS((σ"jobID"="jobID"("JOB")⋈"employerID"="employerID""EMPLOYER"))))

Query 2: Union of distinct usernames from "EMPLOYER" and "APPLICANT":
π"userID"("EMPLOYER") ∪ π"userID"("APPLICANT")−π"userID"("EMPLOYER")∩ π "userID"("APPLICANT")

Query 3: Categorize the number of employers and applicants for each unique username:
π"userID", "Employer", "Count"(σ "Employer"("EMPLOYER"⋈ "userID"="userID" ρ "Employer"(π "userID" ("EMPLOYER"×"EMPLOYER"))) ∪ π "userID", "Applicant","Count"(σ"Applicant" ("APPLICANT"⋈ "userID"="userID" ρ "Applicant"(π "userID"("APPLICANT"×"APPLICANT"))))

Query 4: Average of Salaries of Companies where the average > 70000:
π"company", "Average Salary"(σ AVG("salary") > 70000("JOB")⋈ "jobID"="jobID" "LOCATION")

Query 5: HR employees who have a phone number different from the default value.
π "employeeID", "first_name", "last_name" (σ "phoneNumber"≠ ′ −1 ′ ("HR_EMPLOYEE") ⋈"employeeID"="employeeID" "HR_EMPLOYEE")

Entity: HR_EMPLOYEE

HR_EMPLOYEE(<u>employeeID</u>, first_name, last_name, phoneNumber)

Functional Dependencies:

{employeeID} → {first_name, last_name, phoneNumber}

Entity: USER

USER(<u>userID</u>, username, password)

Functional Dependencies:

{userID} → {username, password}

Entity: EMPLOYER

EMPLOYER(<u>employerID</u>, userID, first_name, last_name, age)

Functional Dependencies:

{employerID} → {userID, first_name, last_name, age}

Entity: ADMINISTRATOR

ADMINISTRATOR(<u>administratorID</u>, userID, first_name, last_name)

Functional Dependencies:

{administratorID} → {userID, first_name, last_name}

Entity: APPLICANT

APPLICANT(<u>applicantID</u>, userID, first_name, last_name, age)

Functional Dependencies:

{applicantID} → {userID, first_name, last_name, age}

Entity: JOB

JOB(<u>jobID</u>, employerID, company, profession, salary, date_posted, deadline_date, duration, arrangement)

Functional Dependencies:

{jobID} → {employerID, company, profession, salary, date_posted, deadline_date, duration, arrangement}

Entity: ADDRESS

ADDRESS(addressID, number, street, zip_code)

Functional Dependencies:

{addressID} → {number, street, zip_code}

Entity: LOCATION

LOCATION(locationID, jobID, addressID)

Functional Dependencies:

{locationID} → {jobID, addressID}

Entity: JOB_QUALIFICATIONS

JOB_QUALIFICATIONS(job_qualificationsID, jobID, experienceLevel, educationLevel)

Functional Dependencies:

{job_qualificationsID} → {jobID, experienceLevel, educationLevel}

Entity: RESUME

RESUME(resumeID, applicantID)

Functional Dependencies:

{resumeID} → {applicantID}

Entity: APPLICANT_QUALIFICATIONS

APPLICANT_QUALIFICATIONS(applicant_qualificationsID, resumeID, experienceLevel, educationLevel)

Functional Dependencies:

{applicant_qualificationsID} → {resumeID, experienceLevel, educationLevel}

Entity: APPLICATION

APPLICATION(applicationID, applicantID, resumeID, status, submissionDate)

Functional Dependencies:

{applicationID} → {applicantID, resumeID, status, submissionDate}

Assignment 7

Here listed below are the explanations for each table on why it is already in 3NF form using Bernstein's Algorithm.

HR_EMPLOYEE TABLE:
```
CREATE TABLE "HR_EMPLOYEE"
(
  "employeeID" SMALLINT PRIMARY KEY, --Primary Key
  "first_name" VARCHAR2(20) NOT NULL,
  "last_name" VARCHAR2(20) NOT NULL,
  "phoneNumber" VARCHAR2(12) DEFAULT '-1' UNIQUE
);
```

The "HR_EMPLOYEE" table is in 3NF because it has a single primary key, "employeeID," which uniquely identifies each record. The attributes "first_name," "last_name," and "phoneNumber" are all fully functionally dependent on the primary key, and there are no transitive dependencies present. Each employee's information is uniquely determined by their employee ID, satisfying the requirements of 3NF.

USER TABLE:
```
CREATE TABLE "USER"
(
  "userID" SMALLINT PRIMARY KEY, --Primary Key
  "username" VARCHAR2(20) NOT NULL UNIQUE,
  "password" VARCHAR2(20) NOT NULL
);
```

The "USER" table is in 3NF as it has a single primary key, "userID," and all non-prime attributes, such as "username" and "password," are fully functionally dependent on this primary key. There are no transitive dependencies, and each user's information is uniquely identified by their user ID, meeting the criteria of 3NF.

EMPLOYER TABLE:
```
CREATE TABLE "EMPLOYER"
(
  "employerID" SMALLINT PRIMARY KEY, --Primary Key
  "userID" SMALLINT REFERENCES "USER"("userID"), --Foreign Key
  "first_name" VARCHAR2(20) NOT NULL,
  "last_name" VARCHAR2(20) NOT NULL,
  "age" SMALLINT DEFAULT '-1');
```

The "EMPLOYER" table is in 3NF because it has a single primary key, "employerID," and all non-prime attributes ("userID," "first_name," "last_name," and "age") are fully functionally dependent on the primary key. The "userID" establishes a foreign key relationship with the "USER" table, and there are no transitive dependencies present, ensuring 3NF compliance.

ADMINISTRATOR TABLE:

```
CREATE TABLE "ADMINISTRATOR"
(
  "administratorID" SMALLINT PRIMARY KEY, --Primary Key
  "userID" SMALLINT REFERENCES "USER"("userID"), --Foreign Key
  "first_name" VARCHAR2(20) NOT NULL,
  "last_name" VARCHAR2(20) NOT NULL
);
```

The "ADMINISTRATOR" table is in 3NF with a primary key, "administratorID," and non-prime attributes ("userID," "first_name," and "last_name") that are fully functionally dependent on this key. The "userID" forms a foreign key relationship with the "USER" table, and there are no transitive dependencies, satisfying the conditions for 3NF.

APPLICANT TABLE:

```
CREATE TABLE "APPLICANT"
(
  "applicantID" SMALLINT PRIMARY KEY, --Primary Key
  "userID" SMALLINT REFERENCES "USER"("userID"), --Foreign Key
  "first_name" VARCHAR2(20) NOT NULL,
  "last_name" VARCHAR2(20) NOT NULL,
  "age" SMALLINT DEFAULT '-1'
);
```

The "APPLICANT" table adheres to 3NF as it possesses a primary key, "applicantID," and all non-prime attributes ("userID," "first_name," "last_name," and "age") are fully functionally dependent on this key. The "userID" establishes a foreign key relationship with the "USER" table, and there are no transitive dependencies, meeting the requirements of 3NF.

JOB TABLE:

```
CREATE TABLE "JOB"
(
  "jobID" SMALLINT PRIMARY KEY, --Primary Key
  "employerID" SMALLINT REFERENCES "EMPLOYER"("employerID"),
--Foreign Key
  "company" VARCHAR2(40) NOT NULL,
  "profession" VARCHAR2(40) NOT NULL,
  "salary" INTEGER NOT NULL,
  "date_posted" DATE NOT NULL,
  "deadline_date" DATE NOT NULL,
  "duration" VARCHAR2(20) DEFAULT 'Unknown',
  "arrangement" VARCHAR2(20) DEFAULT 'Unknown'
);
```

The "JOB" table is in 3NF with a primary key, "jobID," and all non-prime attributes ("employerID," "company," "profession," "salary," "date_posted," "deadline_date," "duration," and "arrangement") being fully functionally dependent on this key. The "employerID" forms a foreign key relationship with the "EMPLOYER" table, and there are no transitive dependencies, ensuring 3NF compliance.

LOCATION TABLE:

```
CREATE TABLE "LOCATION"
(
  "locationID" SMALLINT PRIMARY KEY, --Primary Key
  "jobID" SMALLINT REFERENCES "JOB"("jobID"), --Foreign Key
  "number" SMALLINT DEFAULT '-1',
  "street" VARCHAR2(40) DEFAULT 'N/A',
  "zip_code" VARCHAR2(7) DEFAULT 'N/A'
);
```

The "LOCATION" table is in 3NF as it has a primary key, "locationID," and all non-prime attributes ("jobID," "number," "street," and "zip_code") are fully functionally dependent on this key. The "jobID" establishes a foreign key relationship with the "JOB" table, and there are no transitive dependencies, meeting the criteria of 3NF.

JOB_QUALIFICATIONS TABLE:

```
CREATE TABLE "JOB_QUALIFICATIONS"
(
  "job_qualificationsID" SMALLINT PRIMARY KEY, --Primary Key
  "jobID" SMALLINT REFERENCES "JOB"("jobID"), --Foreign Key
  "experienceLevel" VARCHAR2(40) DEFAULT 'N/A',
  "educationLevel" VARCHAR2(40) DEFAULT 'N/A'
);
```

The "JOB_QUALIFICATIONS" table is in 3NF with a primary key, "job_qualificationsID," and non-prime attributes ("jobID," "experienceLevel," and "educationLevel") that are fully functionally dependent on this key. The "jobID" forms a foreign key relationship with the "JOB" table, and there are no transitive dependencies, satisfying the conditions for 3NF.

RESUME TABLE:

```
CREATE TABLE "RESUME"
(
  "resumeID" SMALLINT PRIMARY KEY, --Primary Key
  "applicantID" SMALLINT REFERENCES "APPLICANT"("applicantID")
--Foreign Key
);
```

The "RESUME" table adheres to 3NF as it possesses a primary key, "resumeID," and the only non-prime attribute, "applicantID," is fully functionally dependent on this key. The "applicantID" establishes a foreign key relationship with the "APPLICANT" table, and there are no transitive dependencies, meeting the requirements of 3NF.

APPLICANT_QUALIFICATIONS TABLE:

```
CREATE TABLE "APPLICANT_QUALIFICATIONS"
(
  "applicant_qualificationsID" SMALLINT PRIMARY KEY, --Primary
Key
  "resumeID" SMALLINT REFERENCES "RESUME"("resumeID"), --Foreign
Key
  "experienceLevel" VARCHAR2(40) DEFAULT 'N/A',
  "educationLevel" VARCHAR2(40) DEFAULT 'N/A'
);
```

The "APPLICANT_QUALIFICATIONS" table is in 3NF with a primary key, "applicant_qualificationsID," and non-prime attributes ("resumeID," "experienceLevel," and "educationLevel") that are fully functionally dependent on this key. The "resumeID" forms a foreign key relationship with the "RESUME" table, and there are no transitive dependencies, satisfying the conditions for 3NF.

APPLICATION TABLE:

```
CREATE TABLE "APPLICATION"
(
  "applicationID" SMALLINT PRIMARY KEY, --Primary Key
  "applicantID" SMALLINT REFERENCES "APPLICANT"("applicantID"),
--Foreign Key
  "resumeID" SMALLINT REFERENCES "RESUME"("resumeID"), --Foreign
Key
  "status" VARCHAR2(20) NOT NULL,
  "submissionDate" DATE NOT NULL
);
```

The "APPLICATION" table is in 3NF as it has a primary key, "applicationID," and non-prime attributes ("applicantID," "resumeID," "status," and "submissionDate") that are fully functionally dependent on this key. The "applicantID" and "resumeID" establish foreign key relationships with the "APPLICANT" and "RESUME" tables, respectively, and there are no transitive dependencies, meeting the criteria of 3NF.

In conclusion, after a thorough examination of the provided database tables, it has been determined that none of the source code required modification. All tables were found to be already in the Third Normal Form (3NF), ensuring data integrity and eliminating redundancy. The confirmation of 3NF compliance was achieved through the application of Bernstein's algorithm.

Bernstein's algorithm involves the identification of functional dependencies within each table and the subsequent examination for transitive dependencies. By systematically analyzing the relationships between attributes and ensuring that all non-prime attributes are fully functionally dependent on the primary key, we were able to confirm the normalization level of each table. In this case, the absence of transitive dependencies in all tables confirmed that they were already in 3NF.

Assignment 8: BCNF

To check if these tables are in Boyce-Codd Normal Form (BCNF), we need to ensure that they meet the following criteria:

- Every determinant (i.e., attribute that uniquely determines another attribute) is a candidate key.
- There are no non-trivial functional dependencies on a candidate key.

Table Analysis with Functional Dependencies:

HR_EMPLOYEE:

- No non-trivial functional dependencies other than the candidate key "**employeeID**" –> {"first_name", "last_name", "phoneNumber"}.

USER:

- No non-trivial functional dependencies other than the candidate key "**userID**" –> {"username", "password"}.

EMPLOYER:

- No non-trivial functional dependencies other than the candidate key "**employerID**" –> {"userID", "first_name", "last_name", "age"}.

ADMINISTRATOR:

- No non-trivial functional dependencies other than the candidate key "**administratorID**" -> {"userID", "first_name", "last_name"}.

APPLICANT:

- No non-trivial functional dependencies other than the candidate key "**applicantID**" -> {"userID", "first_name", "last_name", "age"}.

JOB:

- No non-trivial functional dependencies other than the candidate key "**jobID**" –> {"employerID", "company", "profession", "salary", "date_posted", "deadline_date", "duration", "arrangement"}.

LOCATION:

- No non-trivial functional dependencies other than the candidate key "**locationID**" –> {"jobID", "number", "street", "zip_code"}.

JOB_QUALIFICATIONS:

- No non-trivial functional dependencies other than the candidate key "**job_qualificationsID**" –> {"jobID", "experienceLevel", "educationLevel"}.

RESUME:

- No non-trivial functional dependencies other than the candidate key "**resumeID**" –> {"applicantID"}.

APPLICANT_QUALIFICATIONS:

- No non-trivial functional dependencies other than the candidate key "**applicant_qualificationsID**" –> {"resumeID", "experienceLevel", "educationLevel"}.

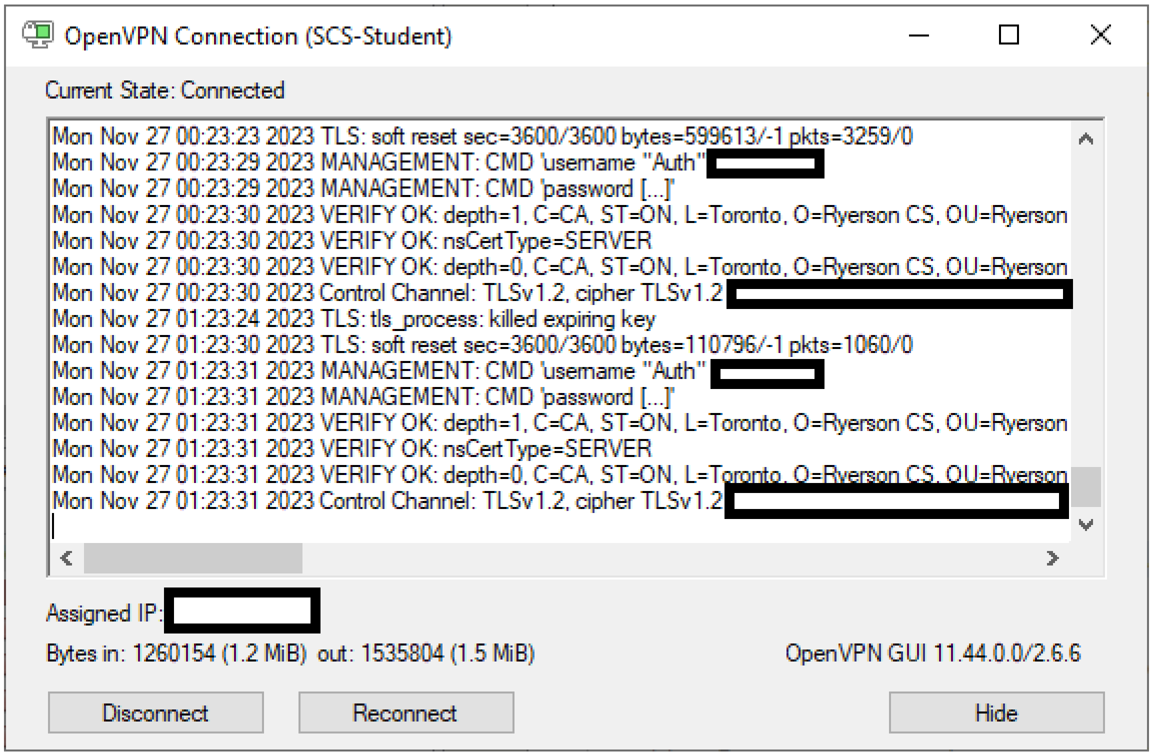
APPLICATION:

- No non-trivial functional dependencies other than the candidate key "**applicationID**" –> {"applicantID", "resumeID", "status", "submissionDate"}.

Based on this analysis, it appears that all the tables are in Boyce-Codd Normal Form (BCNF) as there are no non-trivial functional dependencies on any candidate keys.

ReadMe: How to Run the Job Bank GUI Application

1. Ensure you are on the TMU Comp Sci network (either on campus or at home and connected to the network via OpenVPN)



Current State: Connected

```
Mon Nov 27 00:23:23 2023 TLS: soft reset sec=3600/3600 bytes=599613/-1 pkts=3259/0
Mon Nov 27 00:23:29 2023 MANAGEMENT: CMD 'username "Auth" [redacted]
Mon Nov 27 00:23:29 2023 MANAGEMENT: CMD 'password [...]'
Mon Nov 27 00:23:30 2023 VERIFY OK: depth=1, C=CA, ST=ON, L=Toronto, O=Ryerson CS, OU=Ryerson
Mon Nov 27 00:23:30 2023 VERIFY OK: nsCertType=SERVER
Mon Nov 27 00:23:30 2023 VERIFY OK: depth=0, C=CA, ST=ON, L=Toronto, O=Ryerson CS, OU=Ryerson
Mon Nov 27 00:23:30 2023 Control Channel: TLSv1.2, cipher TLSv1.2 [redacted]
Mon Nov 27 01:23:24 2023 TLS: tls_process: killed expiring key
Mon Nov 27 01:23:30 2023 TLS: soft reset sec=3600/3600 bytes=110796/-1 pkts=1060/0
Mon Nov 27 01:23:31 2023 MANAGEMENT: CMD 'username "Auth" [redacted]
Mon Nov 27 01:23:31 2023 MANAGEMENT: CMD 'password [...]'
Mon Nov 27 01:23:31 2023 VERIFY OK: depth=1, C=CA, ST=ON, L=Toronto, O=Ryerson CS, OU=Ryerson
Mon Nov 27 01:23:31 2023 VERIFY OK: nsCertType=SERVER
Mon Nov 27 01:23:31 2023 VERIFY OK: depth=0, C=CA, ST=ON, L=Toronto, O=Ryerson CS, OU=Ryerson
Mon Nov 27 01:23:31 2023 Control Channel: TLSv1.2, cipher TLSv1.2 [redacted]
```

Assigned IP: [redacted]

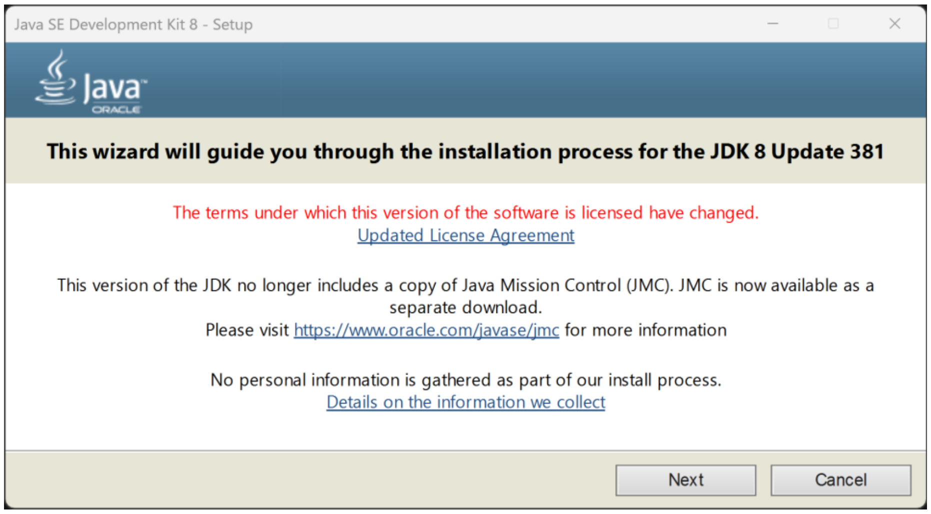
Bytes in: 1260154 (1.2 MiB) out: 1535804 (1.5 MiB)

OpenVPN GUI 11.44.0.0/2.6.6

(For this guide I will use OpenVPN, shown above)

2. Download and install the following software:
   - Java SE Development Kit 8 (Our group used version 8u381, obtained from here: https://www.oracle.com/java/technologies/javase/javase8u211-later-archive-downloads.html NOTE: An Oracle account is required to download)

     Image of Java SE Development Kit 8 Setup (use all default options):



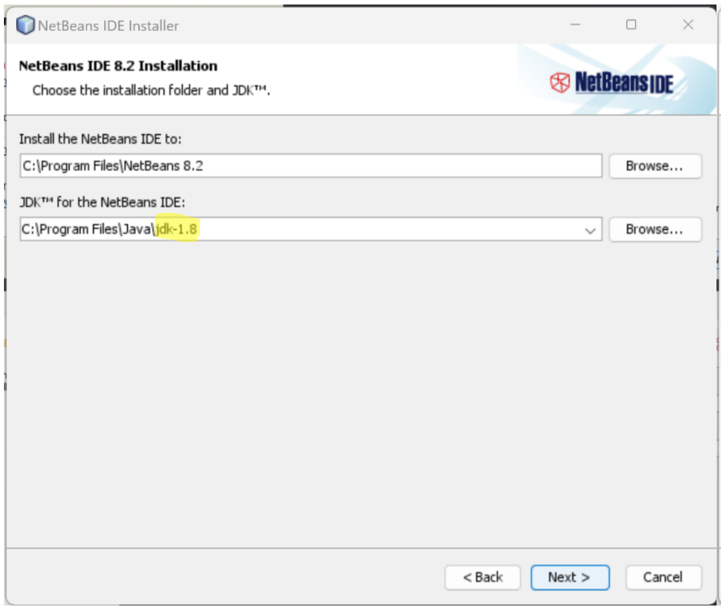
This wizard will guide you through the installation process for the JDK 8 Update 381

The terms under which this version of the software is licensed have changed.
Updated License Agreement

This version of the JDK no longer includes a copy of Java Mission Control (JMC). JMC is now available as a separate download.
Please visit https://www.oracle.com/javase/jmc for more information

No personal information is gathered as part of our install process.
Details on the information we collect

- NetBeans 8.2 (Our group obtained this software from here: https://netbeans-ide.informer.com/8.2/). Ensure during the setup that you select “jdk-1.8” for each of the screens below during the setup. The relevant part is highlighted in yellow:



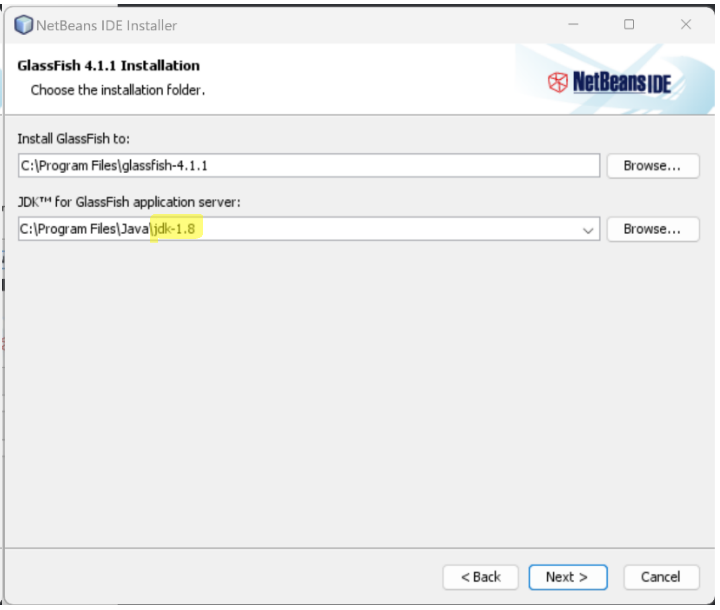
NetBeans IDE Installer

NetBeans IDE 8.2 Installation

Choose the installation folder and JDK™.

Install the NetBeans IDE to: C:\Program Files\NetBeans 8.2

JDK™ for the NetBeans IDE: C:\Program Files\Java\jdk-1.8



NetBeans IDE Installer

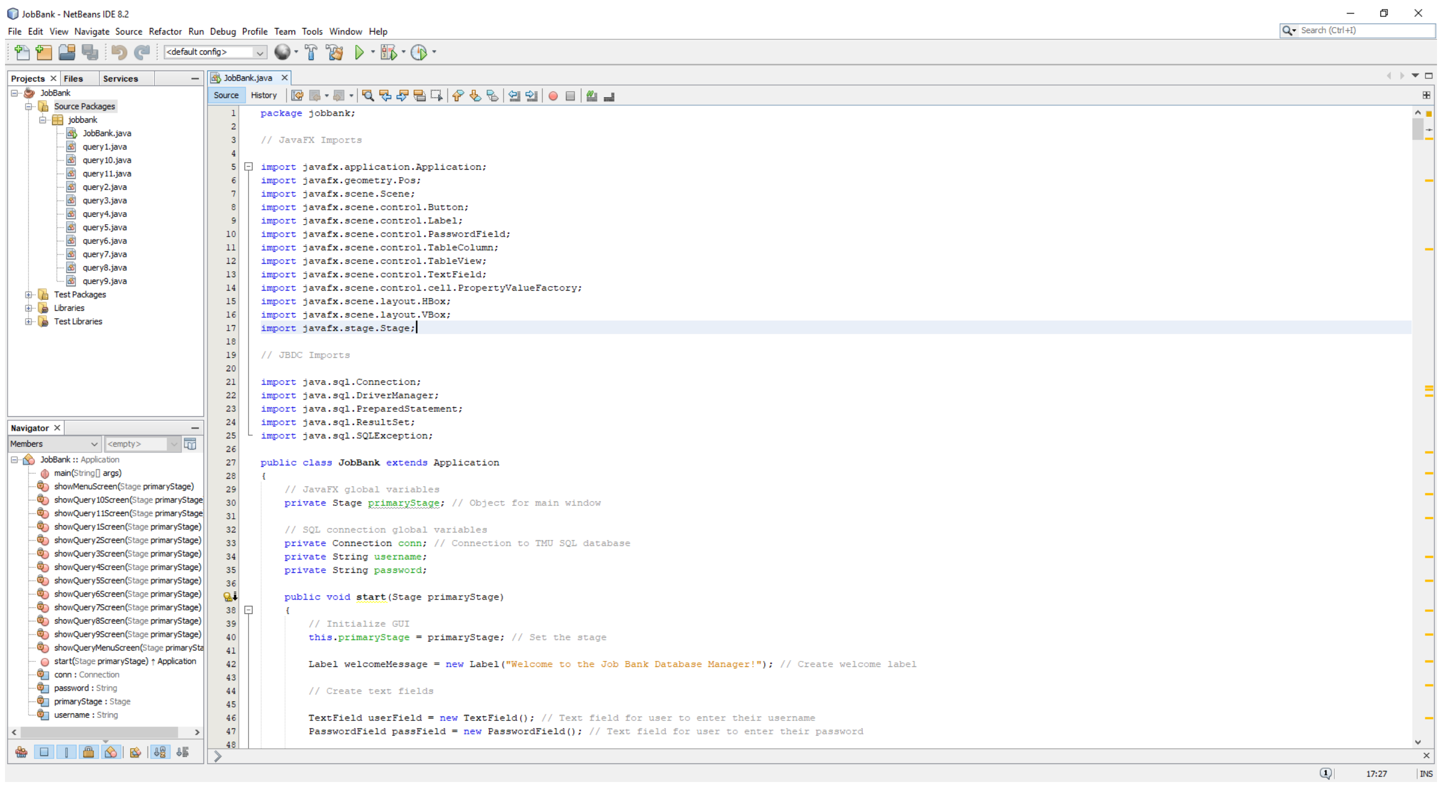
GlassFish 4.1.1 Installation

Choose the installation folder.

Install GlassFish to: C:\Program Files\glassfish-4.1.1

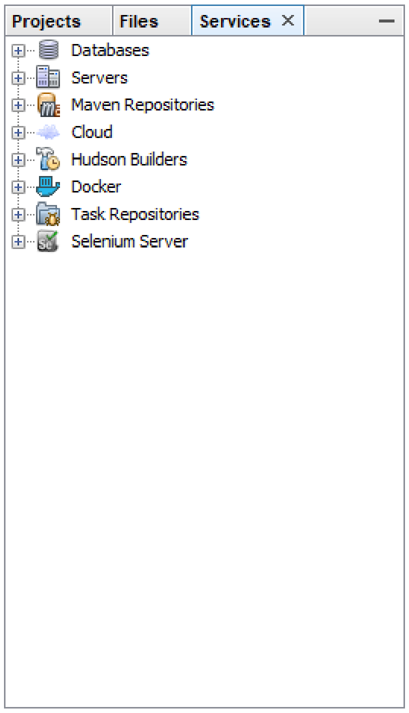
JDK™ for GlassFish application server: C:\Program Files\Java\jdk-1.8

3. Once NetBeans 8.2 is installed, extract and open the attached zip folder as a project in NetBeans. You should see the following screen at this point:



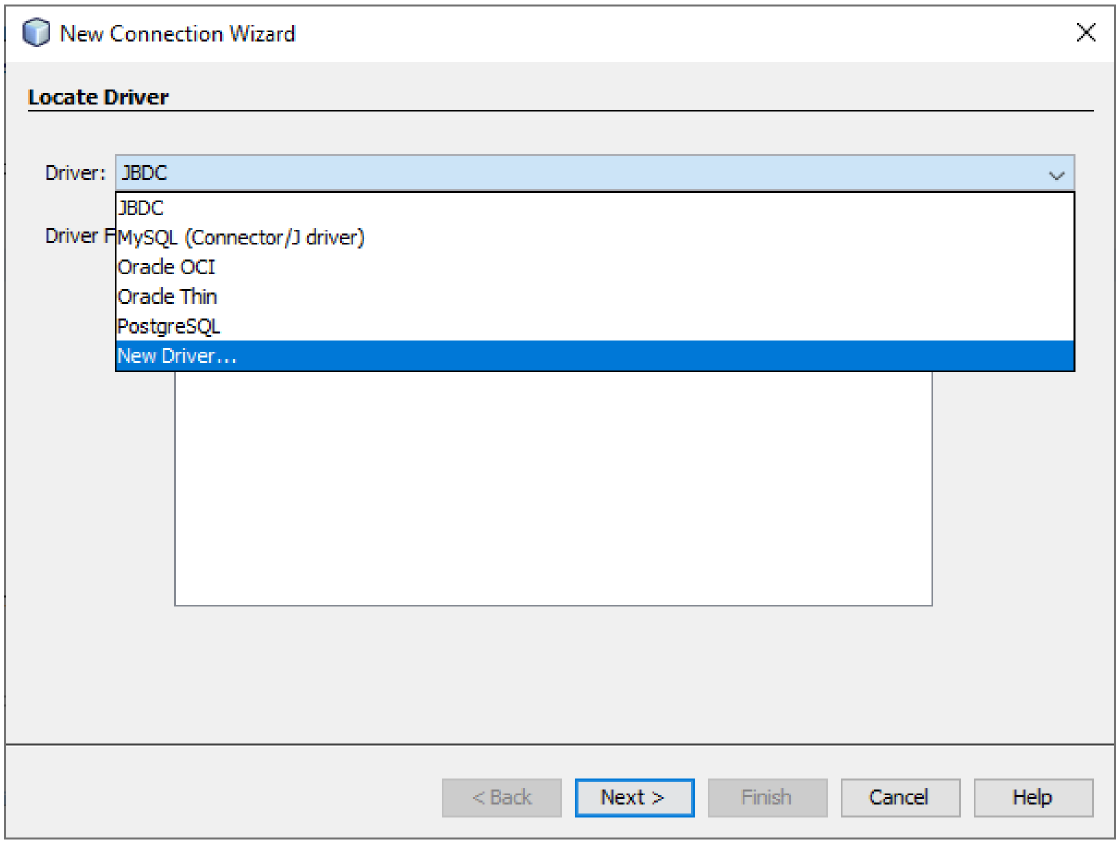
```
package jobbank;

// JavaFX Imports

import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.PasswordField;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

// JBDC Imports

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class JobBank extends Application
{
    // JavaFX global variables
    private Stage primaryStage; // Object for main window

    // SQL connection global variables
    private Connection conn; // Connection to TMU SQL database
    private String username;
    private String password;

    public void start(Stage primaryStage)
    {
        // Initialize GUI
        this.primaryStage = primaryStage; // Set the stage

        Label welcomeMessage = new Label("Welcome to the Job Bank Database Manager!"); // Create welcome label

        // Create text fields

        TextField userField = new TextField(); // Text field for user to enter their username
        PasswordField passField = new PasswordField(); // Text field for user to enter their password
```

4. On the top bar in NetBeans, go to Window > Services. You will then see a panel in the top left of NetBeans. This will not be in a new window, but will be where the project browser usually is. It is shown below:

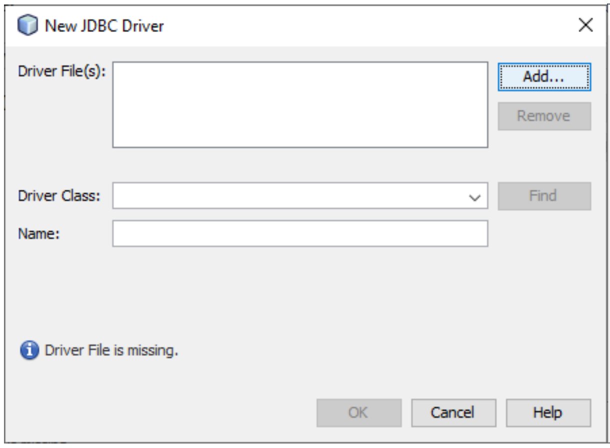


5. Right click “Databases” and click “New Connection”.

6. A “New Connection Wizard” window will appear. Click the drop-down for “Driver” and select “New Driver”, as shown in the image below:



7. After selecting “New Driver”, you will see a window for “New JDBC Driver”. Click the “Add…” button on the top right and navigate to the “/lib” folder of the project. Images are shown below to help:



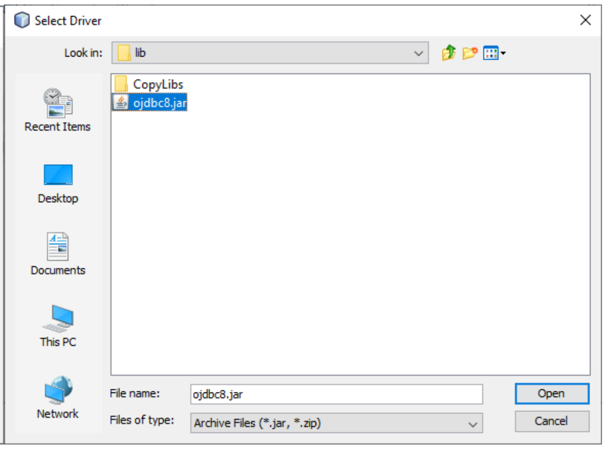
New JDBC Driver

Driver File(s):

Driver Class:

Name:

Driver File is missing.



Select Driver

Look in: lib

CopyLibs
ojdbc8.jar

File name: ojdbc8.jar

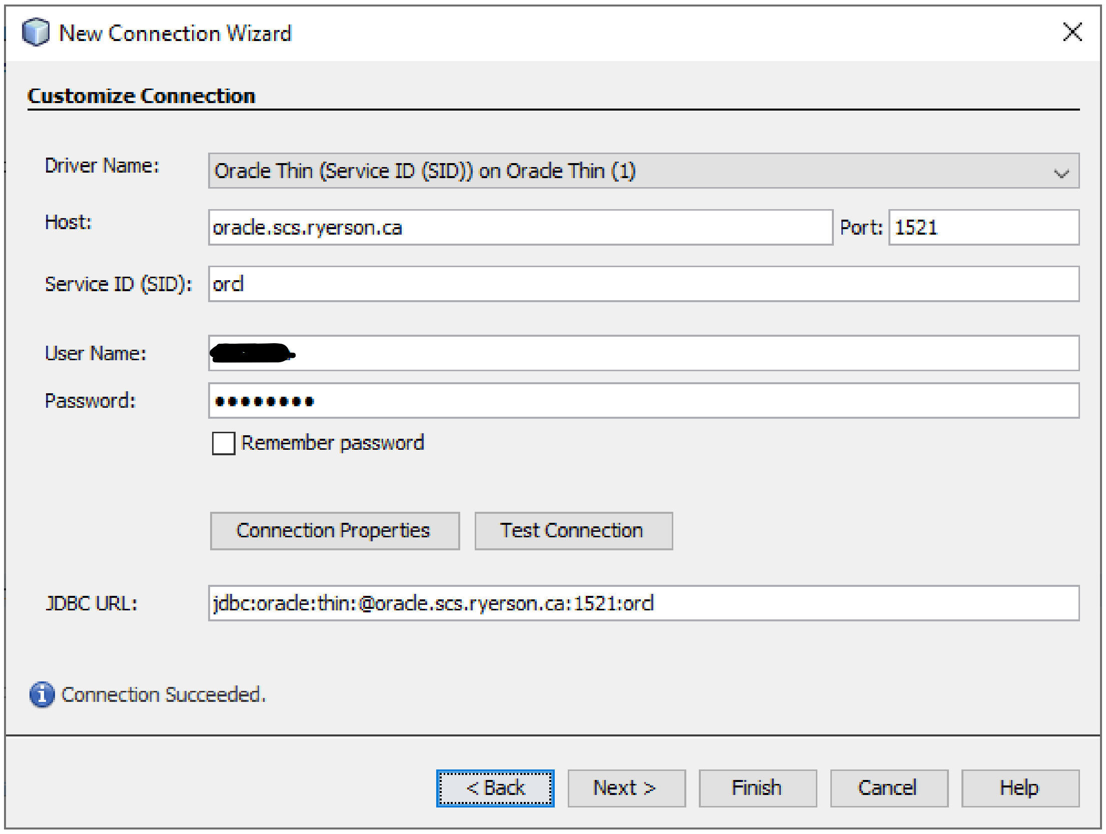
Files of type: Archive Files (*.jar, *.zip)

(Once you see the screen on the left, click “Add…” then navigate to the “/lib” folder in the project folder, shown in the right image)

8. Select the file “ojdbc8.jar” and open. Click “OK” then “Next >”.

9. Here you will see the “Customize Connection” screen. Enter the following information:
   - Host: oracle.scs.ryerson.ca
   - Port: 1521
   - Service ID (SID): orcl
   - User Name: [your TMU short ID]
   - Password: [MMDDXXXX, where MMDD is your birthday and XXXX is last 4 digits of student #]

   An example of this window filled out properly is shown below:



New Connection Wizard

Customize Connection

Driver Name: Oracle Thin (Service ID (SID)) on Oracle Thin (1)

Host: oracle.scs.ryerson.ca Port: 1521

Service ID (SID): orcl

User Name: [redacted]

Password: ••••••••

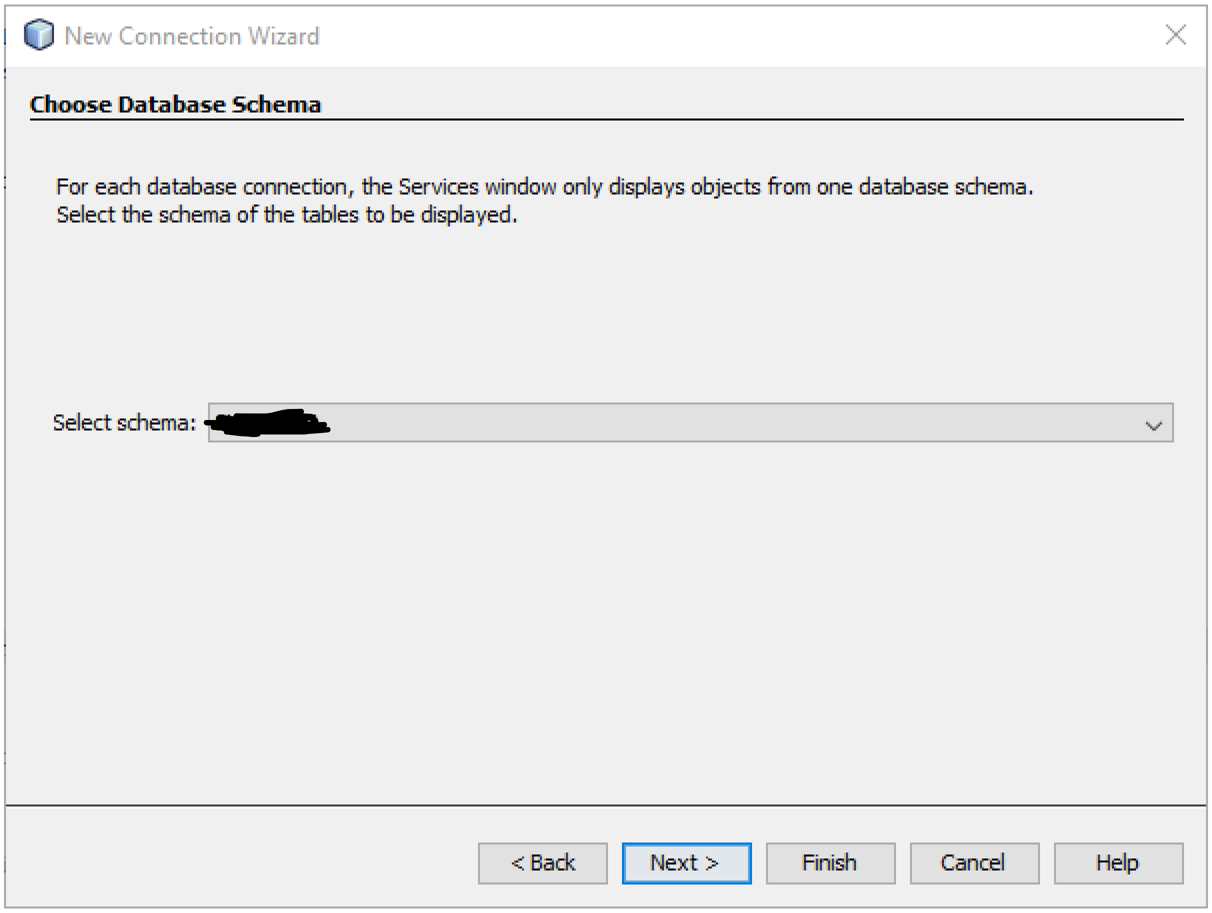
☐ Remember password

JDBC URL: jdbc:oracle:thin:@oracle.scs.ryerson.ca:1521:orcl

Connection Succeeded.

You can press the "Test Connection" button to ensure it was setup successfully. The image above shows a notification in the bottom left stating that the connection was successful.

10. Click "Next" and you will see an option to choose the database schema. Your TMU short ID should be automatically shown here in the drop-down menu. Do not change these settings.

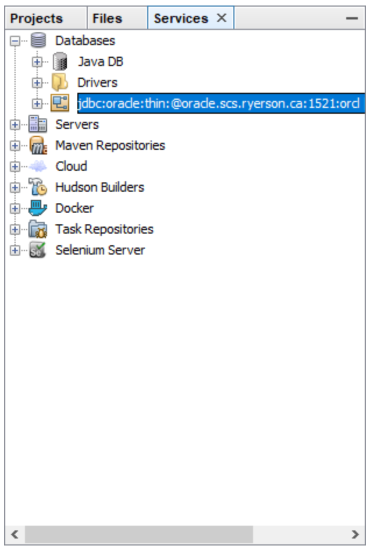


For each database connection, the Services window only displays objects from one database schema.
Select the schema of the tables to be displayed.

Select schema: [redacted]

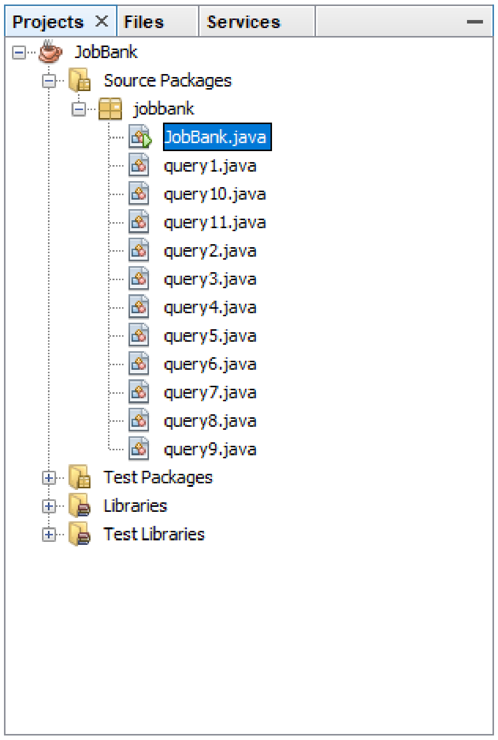
("Choose Database Schema" window)

11. Click "Finish" in the bottom right of the "Choose Database Schema" window. You should now see the connection show up in the "Services" window in the panel shown in step 4:

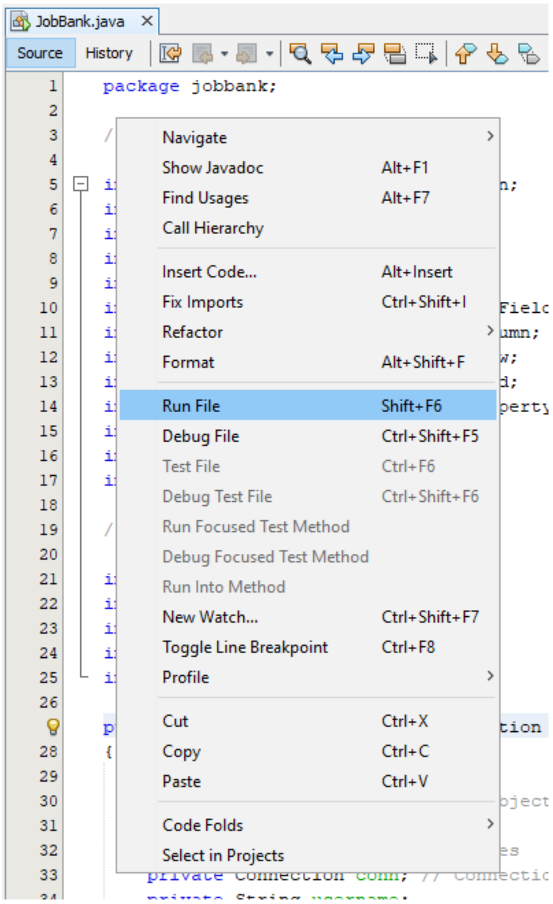


(The image on the left shows the connection highlighted in blue.)

12. In the same panel as step 11, click the “Projects” tab and open “JobBank.java” in the project tree, as shown below:

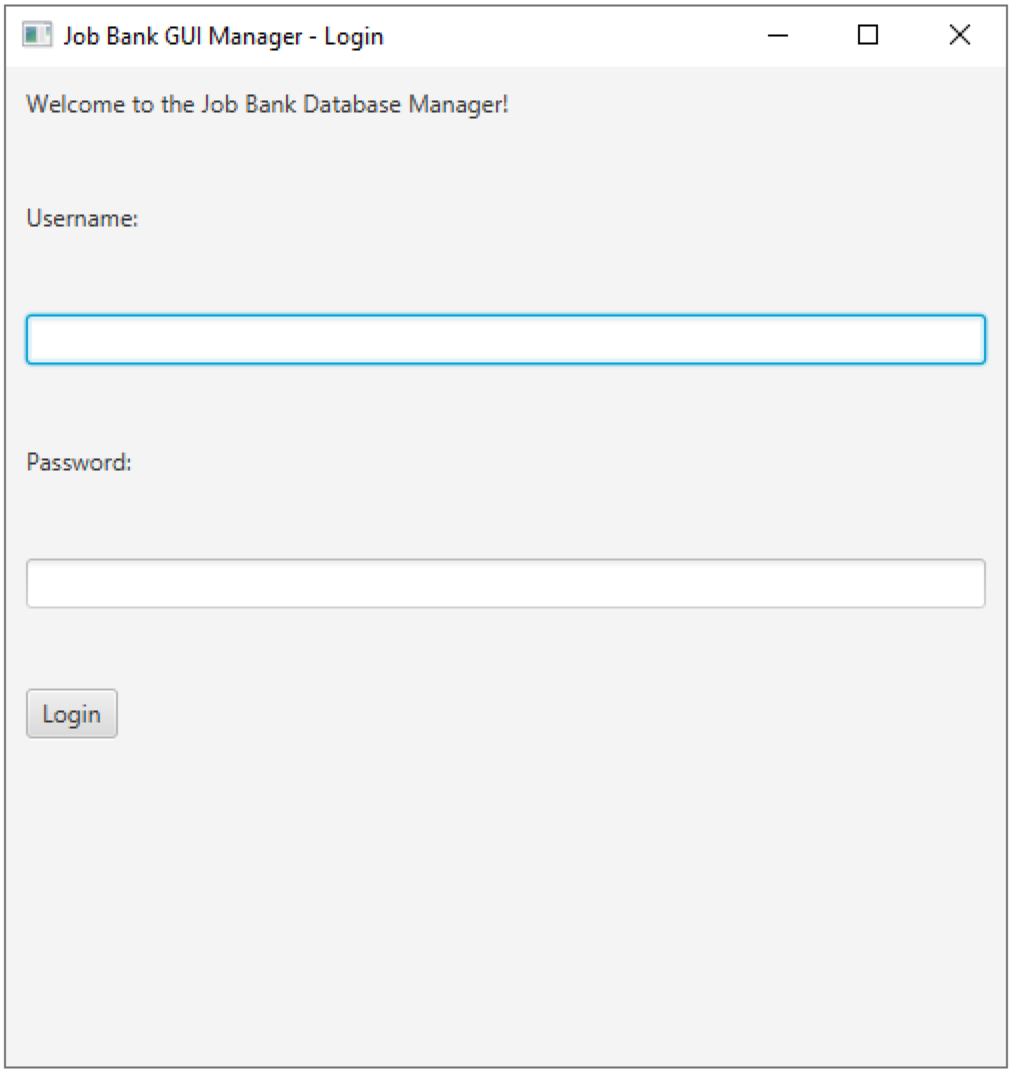


13. Once the code is open in the main window, right click the code and press “Run File”. The code should now run.



```
package jobbank;
```

14. Once the program is running, you will see the login screen:


Job Bank GUI Manager - Login
Welcome to the Job Bank Database Manager!
Username:
Password:
Login

From here, enter the same username and password you used in step 9, that is:

- User Name: [your TMU short ID]
- Password: [MMDDXXXX, where MMDD is your birthday and XXXX is last 4 digits of student #]

You should now be able to use the Job Bank database GUI.

Assignment 9: Job Bank GUI

Section 1: Introduction

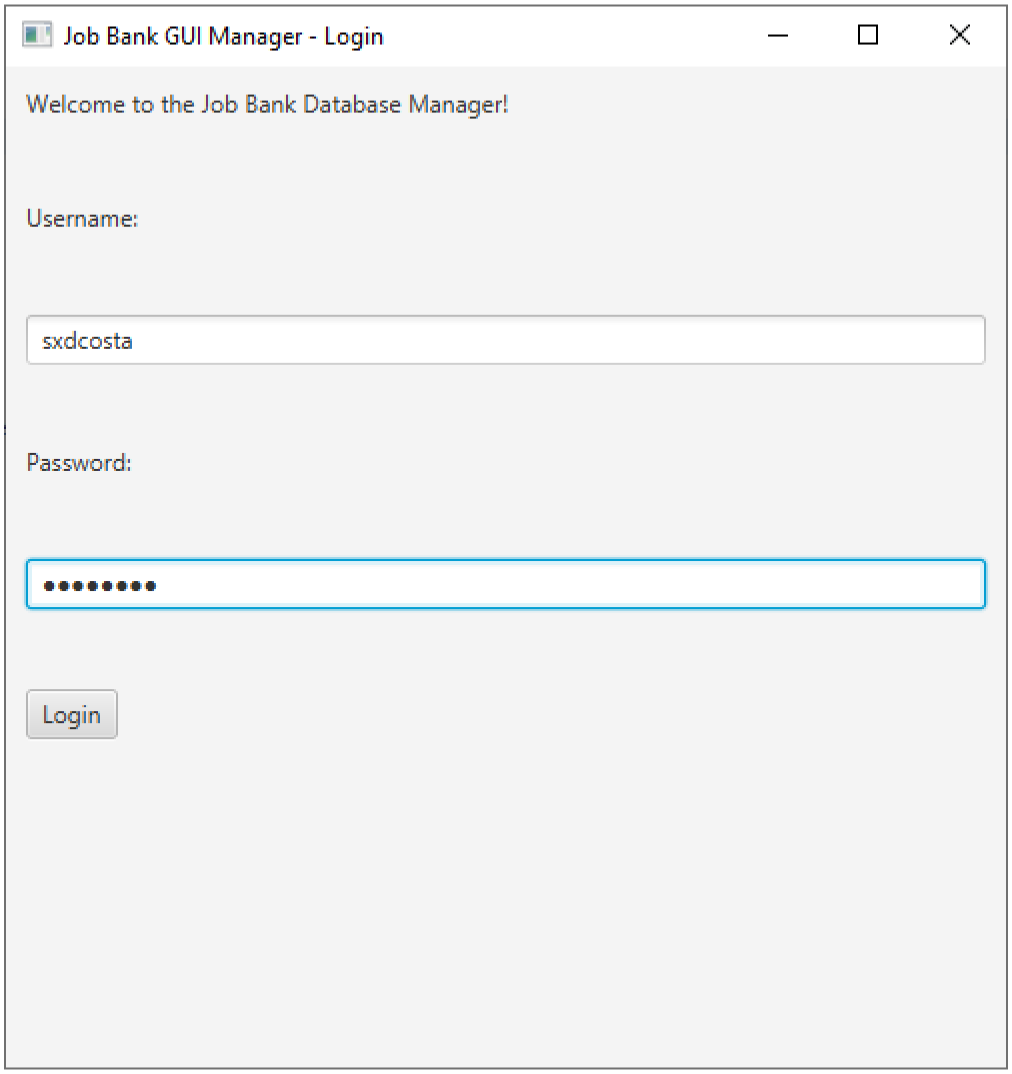
Before starting assignment 9, all parts were verified to be in 3NF and BCNF through assignments 7 and 8. There are no special cases, and the output of the advanced reports (queries) will be shown in the screenshots below via the GUI interface.

For this assignment, we decided to develop the GUI interface in Java using JavaFX to create the UI and JBDC to have our program interface with the TMU SQL database. Screenshots of the program running as well as a brief description of what is happening in each image is shown in section 2 of this report.

Section 2: Setup

To run this application, NetBeans 8.2 was used, as well as JBDC 8. This version of NetBeans was used as it has JavaFX built in, and JBDC 8 was used as it is compatible with Java SDK 8, which this application uses. The JBDC driver is bundled with this project in the “/lib” folder.

Section 3: Demonstration of GUI


Job Bank GUI Manager - Login

Welcome to the Job Bank Database Manager!

Username:

sxdcosta

Password:

•••••••••

Figure 1: Login Screen

Figure 1 shows the login screen of the application, which the user is shown upon launching the program for the first time in a session. For the username and password, the user is required to enter their TMU SQL credentials: that is, their "my.torontomu" short ID and their password for the computer science computers (usually their date of birth in the form of MMDDXXXX, where XXX is the last 4 digits of their student number).

Once logged in, the user is presented with the options menu, shown in figure 2, where the user can select how they want to interact with the database:

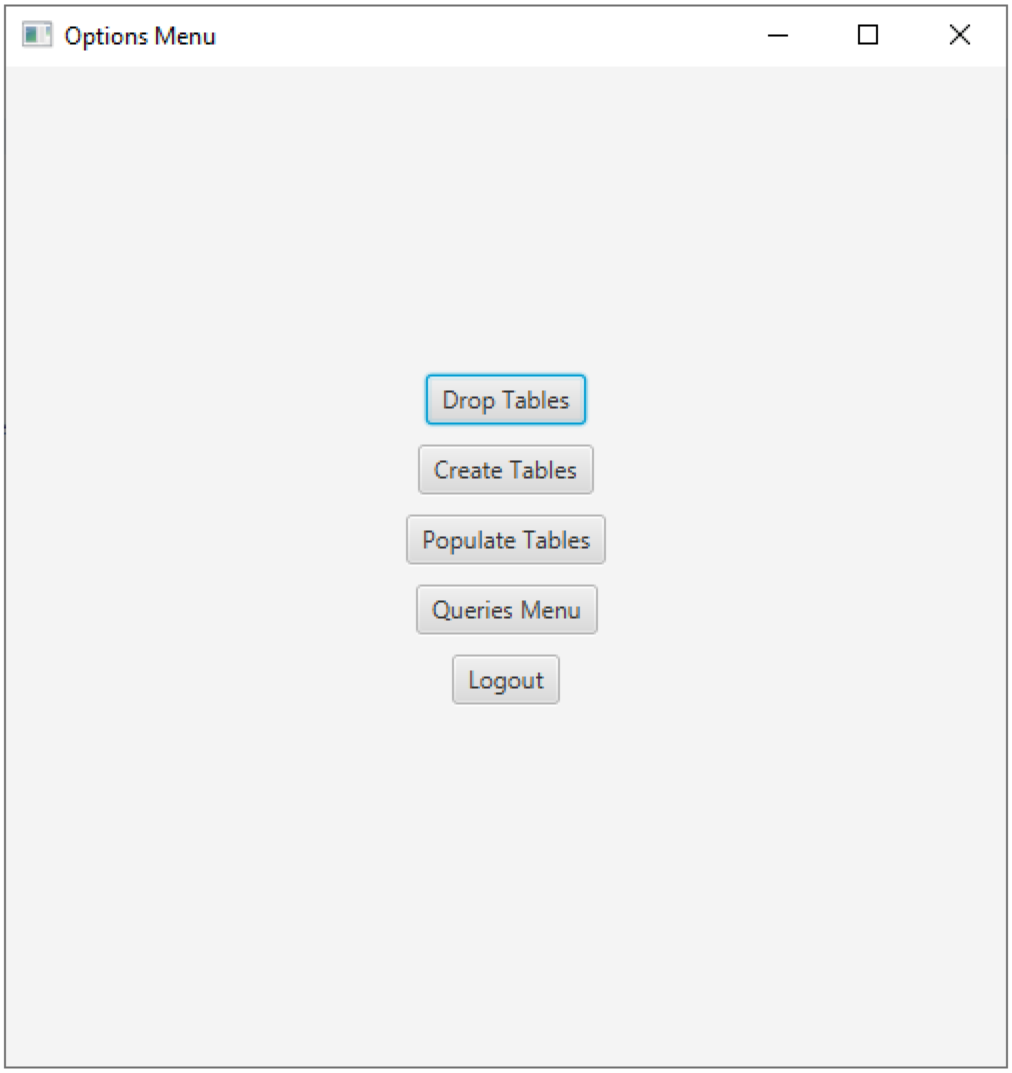
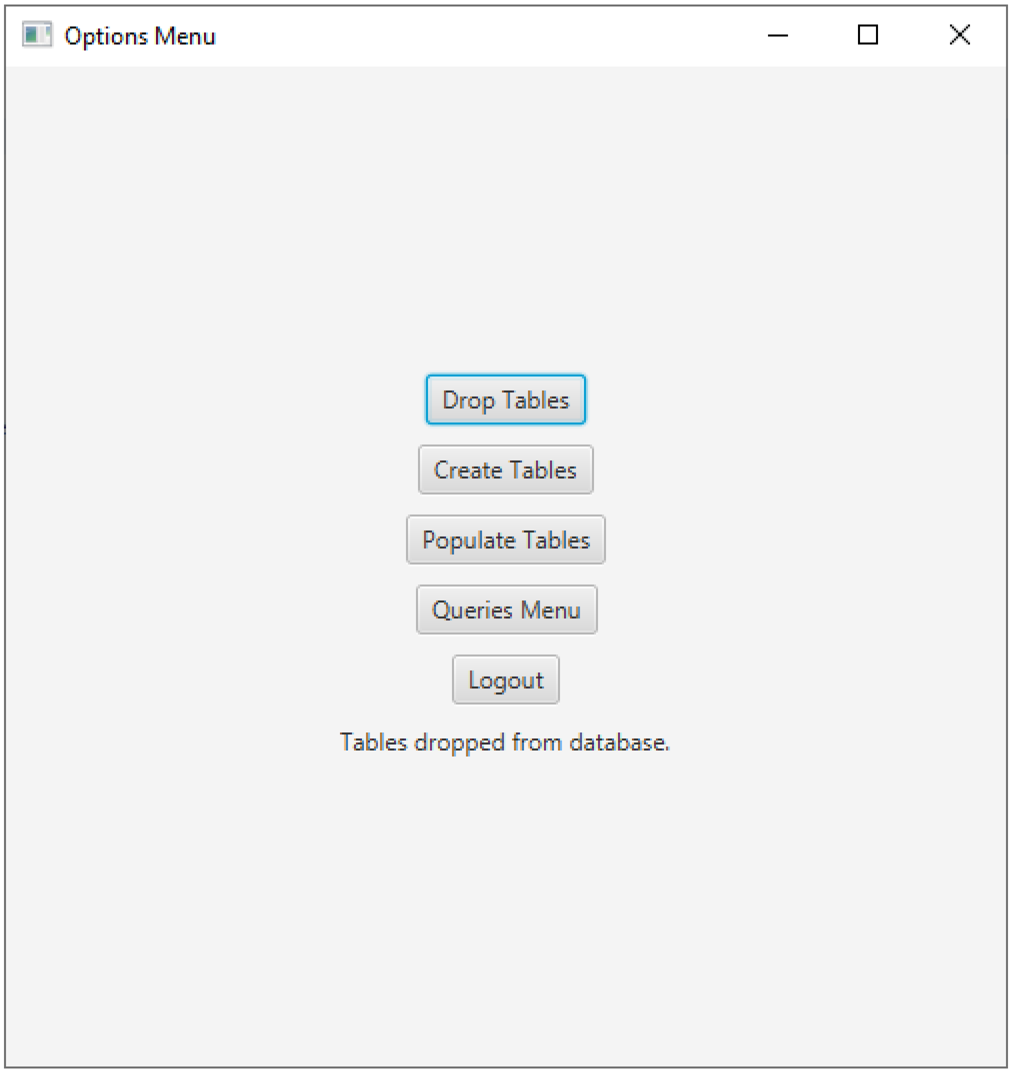


Figure 2: Options Menu

Here, the user can

- Drop Tables: Drop all tables in the database (losing all data)
- Create Tables: Use the schema from previous assignments to recreate the tables with no dummy data
- Populate Tables: Insert dummy data into the newly created tables so that queries can be made
- Queries Menu: See a list of premade queries using the dummy data
- Logout: Securely logout of the database by closing the connection to TMU SQL servers

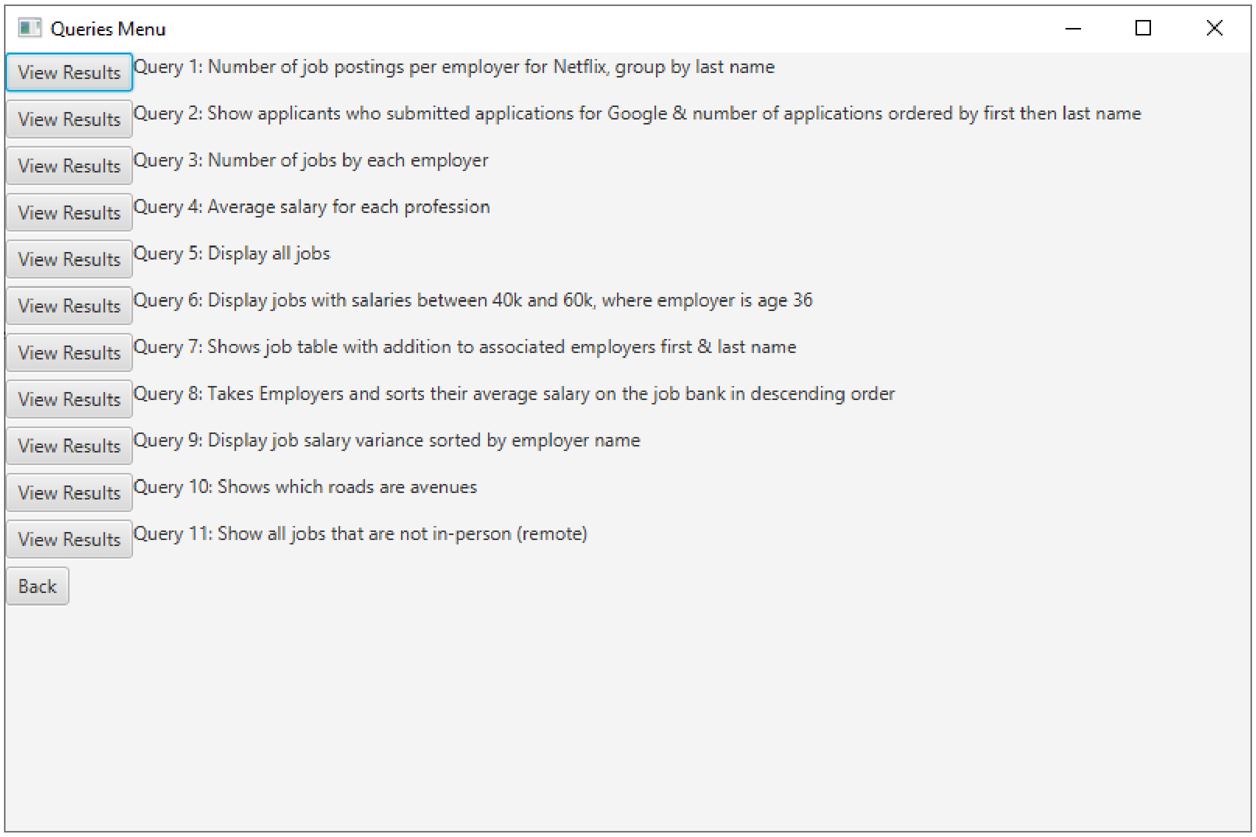
If the user decides to do any of the first 3 options, they will see a notification at the bottom stating that the action was successful, as shown below:



Tables dropped from database.

Figure 3: Notification at the bottom announcing that the action was successful

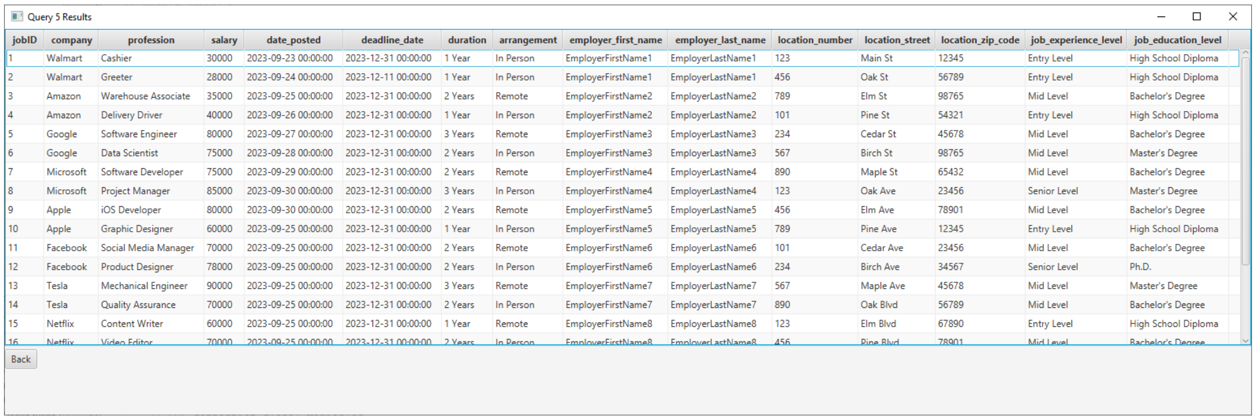
Once the user has populated the database with dummy data, they can then select premade queries from a menu to filter the data in the database, as shown in figure 4:


Queries Menu

View Results Query 1: Number of job postings per employer for Netflix, group by last name
View Results Query 2: Show applicants who submitted applications for Google & number of applications ordered by first then last name
View Results Query 3: Number of jobs by each employer
View Results Query 4: Average salary for each profession
View Results Query 5: Display all jobs
View Results Query 6: Display jobs with salaries between 40k and 60k, where employer is age 36
View Results Query 7: Shows job table with addition to associated employers first & last name
View Results Query 8: Takes Employers and sorts their average salary on the job bank in descending order
View Results Query 9: Display job salary variance sorted by employer name
View Results Query 10: Shows which roads are avenues
View Results Query 11: Show all jobs that are not in-person (remote)
Back

Figure 4: Queries Menu

Here, the user can select a query and view the results presented in a JavaFX table, or return to the options menu in figure 3 by pressing the “Back” button. Selecting query 5, for example, will display all jobs in the database, as shown below:


Query 5 Results

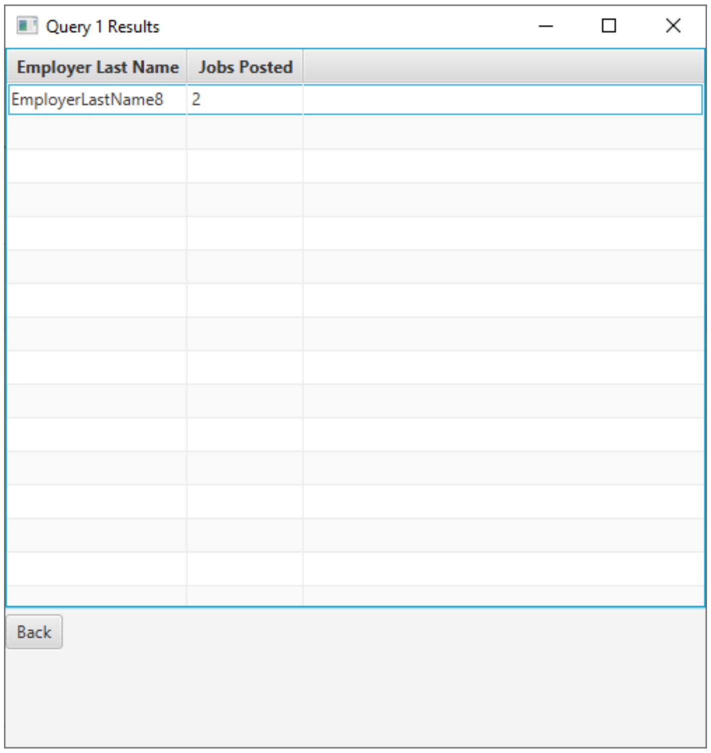
jobID	company	profession	salary	date_posted	deadline_date	duration	arrangement	employer_first_name	employer_last_name	location_number	location_street	location_zip_code	job_experience_level	job_education_level
1	Walmart	Cashier	30000	2023-09-23 00:00:00	2023-12-31 00:00:00	1 Year	In Person	EmployerFirstName1	EmployerLastName1	123	Main St	12345	Entry Level	High School Diploma
2	Walmart	Greeter	28000	2023-09-24 00:00:00	2023-12-11 00:00:00	1 Year	In Person	EmployerFirstName1	EmployerLastName1	456	Oak St	56789	Entry Level	High School Diploma
3	Amazon	Warehouse Associate	35000	2023-09-25 00:00:00	2023-12-31 00:00:00	2 Years	Remote	EmployerFirstName2	EmployerLastName2	789	Elm St	98765	Mid Level	Bachelor's Degree
4	Amazon	Delivery Driver	40000	2023-09-26 00:00:00	2023-12-31 00:00:00	1 Year	In Person	EmployerFirstName2	EmployerLastName2	101	Pine St	54321	Entry Level	High School Diploma
5	Google	Software Engineer	80000	2023-09-27 00:00:00	2023-12-31 00:00:00	3 Years	Remote	EmployerFirstName3	EmployerLastName3	234	Cedar St	45678	Mid Level	Bachelor's Degree
6	Google	Data Scientist	75000	2023-09-28 00:00:00	2023-12-31 00:00:00	2 Years	In Person	EmployerFirstName3	EmployerLastName3	567	Birch St	98765	Mid Level	Master's Degree
7	Microsoft	Software Developer	75000	2023-09-29 00:00:00	2023-12-31 00:00:00	2 Years	Remote	EmployerFirstName4	EmployerLastName4	890	Maple St	65432	Mid Level	Bachelor's Degree
8	Microsoft	Project Manager	85000	2023-09-30 00:00:00	2023-12-31 00:00:00	3 Years	In Person	EmployerFirstName4	EmployerLastName4	123	Oak Ave	23456	Senior Level	Master's Degree
9	Apple	iOS Developer	80000	2023-09-30 00:00:00	2023-12-31 00:00:00	2 Years	Remote	EmployerFirstName5	EmployerLastName5	456	Elm Ave	78901	Mid Level	Bachelor's Degree
10	Apple	Graphic Designer	60000	2023-09-25 00:00:00	2023-12-31 00:00:00	1 Year	In Person	EmployerFirstName5	EmployerLastName5	789	Pine Ave	12345	Entry Level	High School Diploma
11	Facebook	Social Media Manager	70000	2023-09-25 00:00:00	2023-12-31 00:00:00	2 Years	Remote	EmployerFirstName6	EmployerLastName6	101	Cedar Ave	23456	Mid Level	Bachelor's Degree
12	Facebook	Product Designer	78000	2023-09-25 00:00:00	2023-12-31 00:00:00	2 Years	In Person	EmployerFirstName6	EmployerLastName6	234	Birch Ave	34567	Senior Level	Ph.D.
13	Tesla	Mechanical Engineer	90000	2023-09-25 00:00:00	2023-12-31 00:00:00	3 Years	Remote	EmployerFirstName7	EmployerLastName7	567	Maple Ave	45678	Mid Level	Master's Degree
14	Tesla	Quality Assurance	70000	2023-09-25 00:00:00	2023-12-31 00:00:00	2 Years	In Person	EmployerFirstName7	EmployerLastName7	890	Oak Blvd	56789	Mid Level	Bachelor's Degree
15	Netflix	Content Writer	60000	2023-09-25 00:00:00	2023-12-31 00:00:00	1 Year	Remote	EmployerFirstName8	EmployerLastName8	123	Elm Blvd	67890	Entry Level	High School Diploma
16	Netflix	Video Editor	70000	2023-09-25 00:00:00	2023-12-31 00:00:00	2 Years	In Person	EmployerFirstName8	EmployerLastName8	456	Pine Blvd	78901	Mid Level	Bachelor's Degree

Back

Figure 5: Example of viewing a query (query 5: all jobs)

This screen shows all the columns as well as a scrollable bar to see more results that were not able to fit on the screen. There is also a “Back” button to return to the queries menu in figure 4.

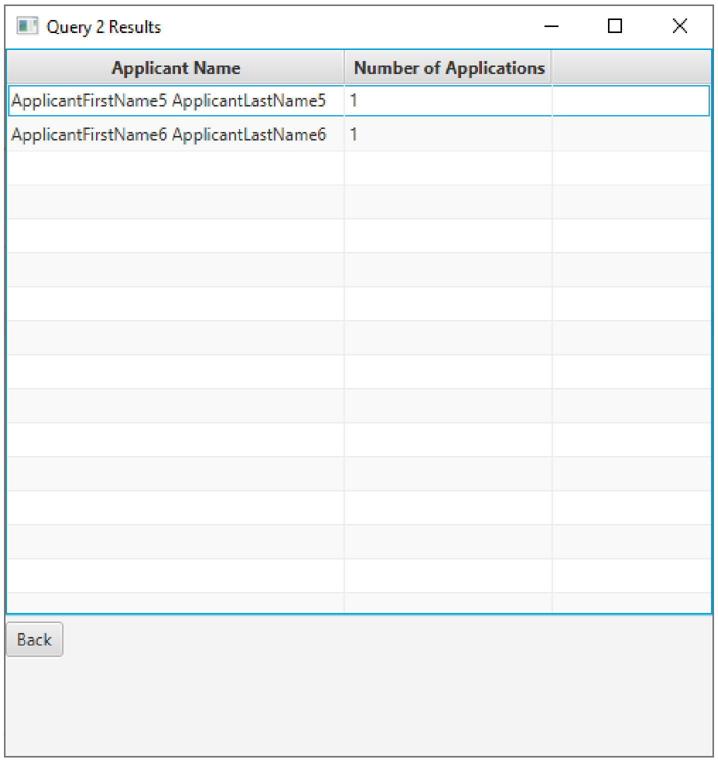
Section 3: Advanced Reports from GUI



Query 1 Results

Employer Last Name	Jobs Posted
EmployerLastName8	2

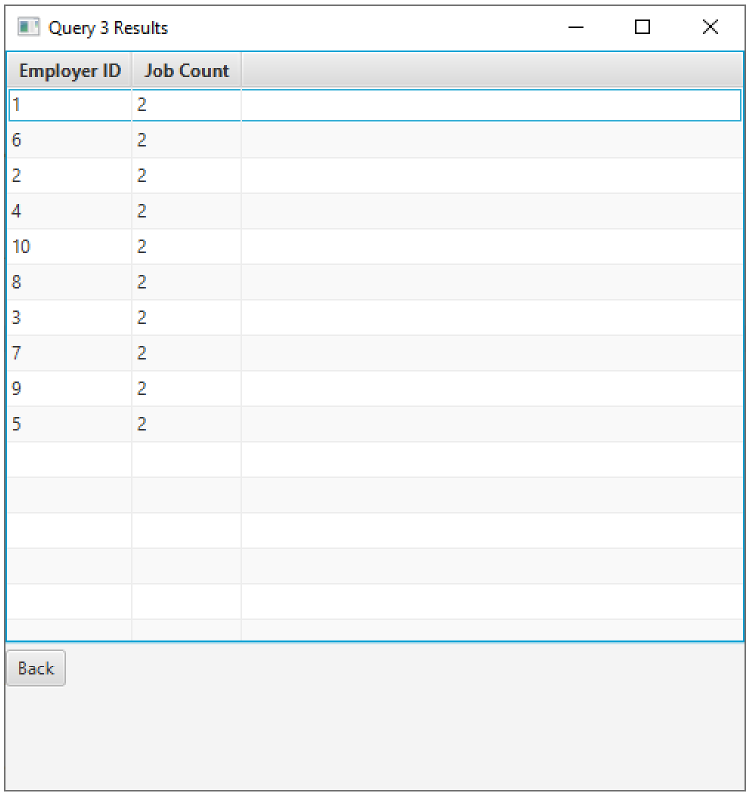
Figure 6: Query 1 Results



Query 2 Results

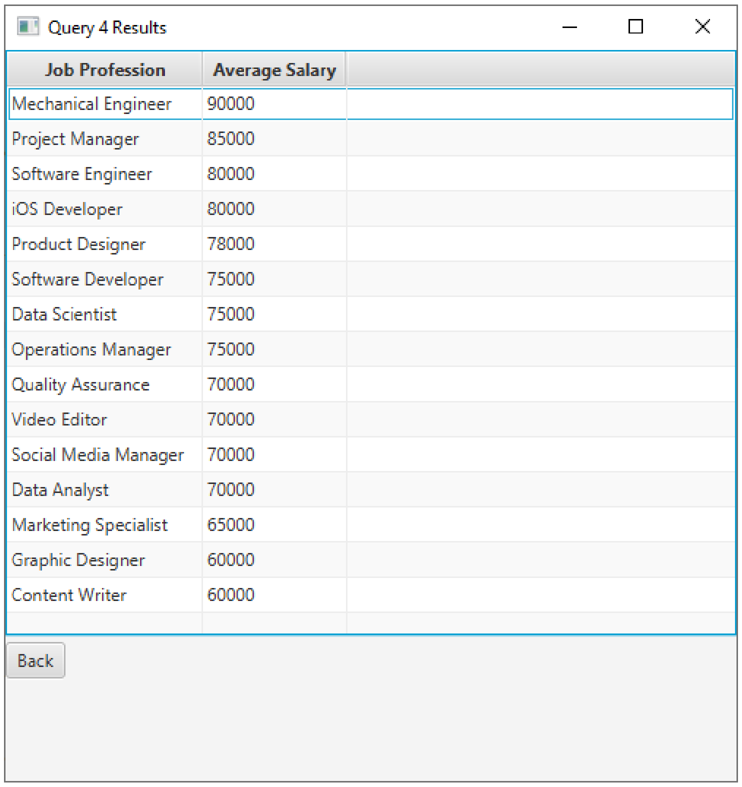
Applicant Name	Number of Applications
ApplicantFirstName5 ApplicantLastName5	1
ApplicantFirstName6 ApplicantLastName6	1

Figure 7: Query 2 Results


Query 3 Results

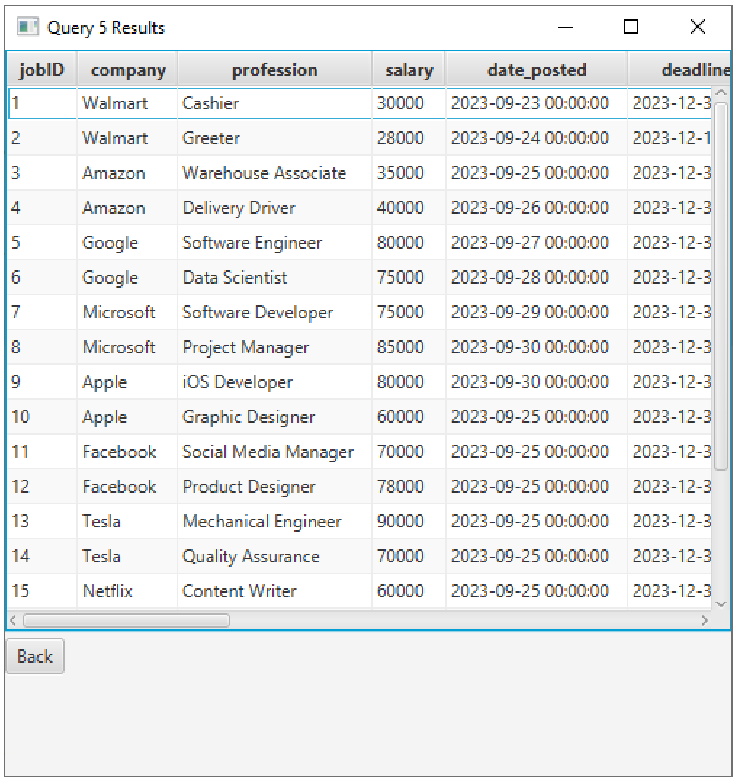
Employer ID	Job Count
1	2
6	2
2	2
4	2
10	2
8	2
3	2
7	2
9	2
5	2

Figure 8: Query 3 Results


Query 4 Results

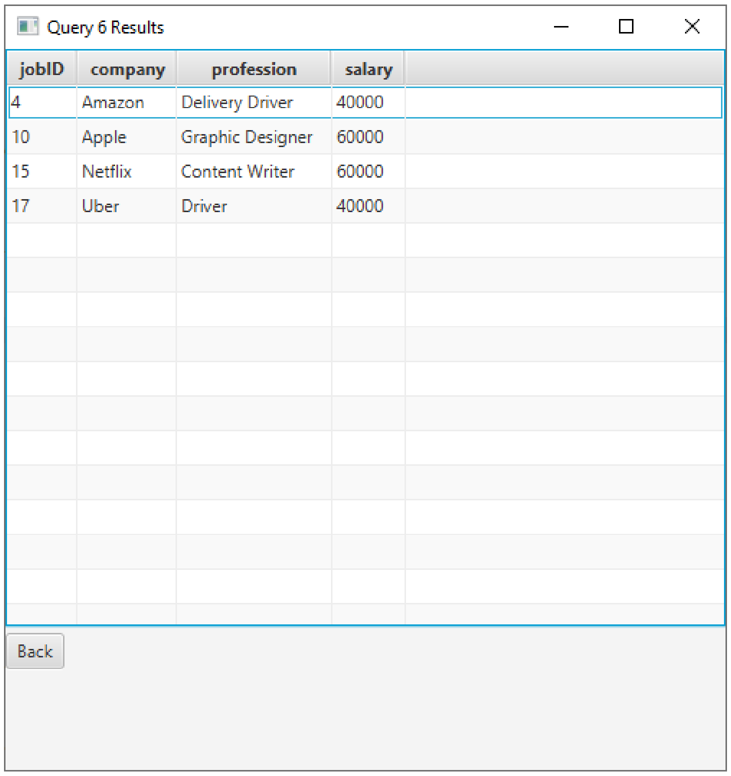
Job Profession	Average Salary
Mechanical Engineer	90000
Project Manager	85000
Software Engineer	80000
iOS Developer	80000
Product Designer	78000
Software Developer	75000
Data Scientist	75000
Operations Manager	75000
Quality Assurance	70000
Video Editor	70000
Social Media Manager	70000
Data Analyst	70000
Marketing Specialist	65000
Graphic Designer	60000
Content Writer	60000

Figure 9: Query 4 Results



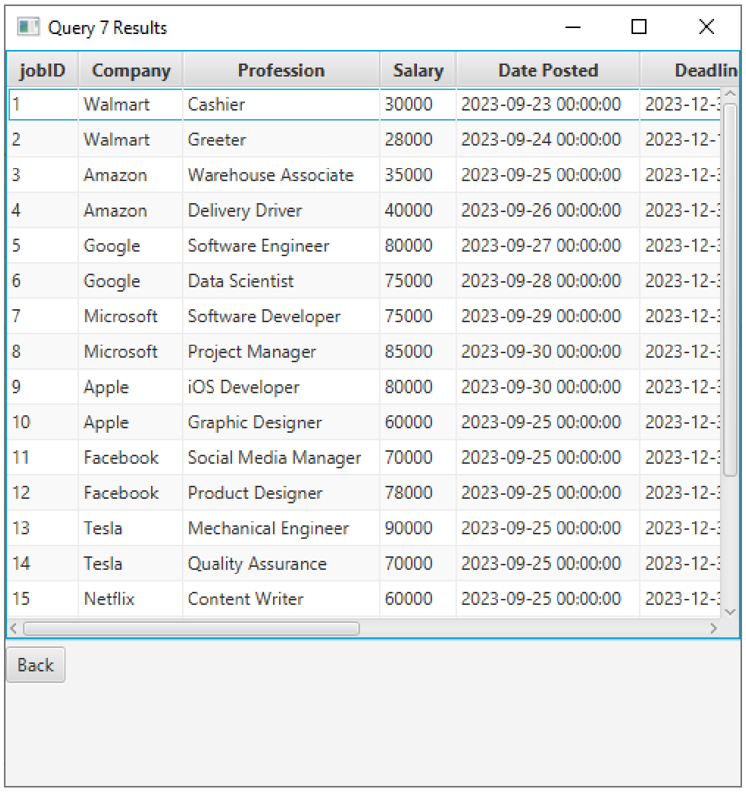
jobID	company	profession	salary	date_posted	deadline
1	Walmart	Cashier	30000	2023-09-23 00:00:00	2023-12-3
2	Walmart	Greeter	28000	2023-09-24 00:00:00	2023-12-1
3	Amazon	Warehouse Associate	35000	2023-09-25 00:00:00	2023-12-3
4	Amazon	Delivery Driver	40000	2023-09-26 00:00:00	2023-12-3
5	Google	Software Engineer	80000	2023-09-27 00:00:00	2023-12-3
6	Google	Data Scientist	75000	2023-09-28 00:00:00	2023-12-3
7	Microsoft	Software Developer	75000	2023-09-29 00:00:00	2023-12-3
8	Microsoft	Project Manager	85000	2023-09-30 00:00:00	2023-12-3
9	Apple	iOS Developer	80000	2023-09-30 00:00:00	2023-12-3
10	Apple	Graphic Designer	60000	2023-09-25 00:00:00	2023-12-3
11	Facebook	Social Media Manager	70000	2023-09-25 00:00:00	2023-12-3
12	Facebook	Product Designer	78000	2023-09-25 00:00:00	2023-12-3
13	Tesla	Mechanical Engineer	90000	2023-09-25 00:00:00	2023-12-3
14	Tesla	Quality Assurance	70000	2023-09-25 00:00:00	2023-12-3
15	Netflix	Content Writer	60000	2023-09-25 00:00:00	2023-12-3

Figure 10: Query 5 Results



jobID	company	profession	salary
4	Amazon	Delivery Driver	40000
10	Apple	Graphic Designer	60000
15	Netflix	Content Writer	60000
17	Uber	Driver	40000

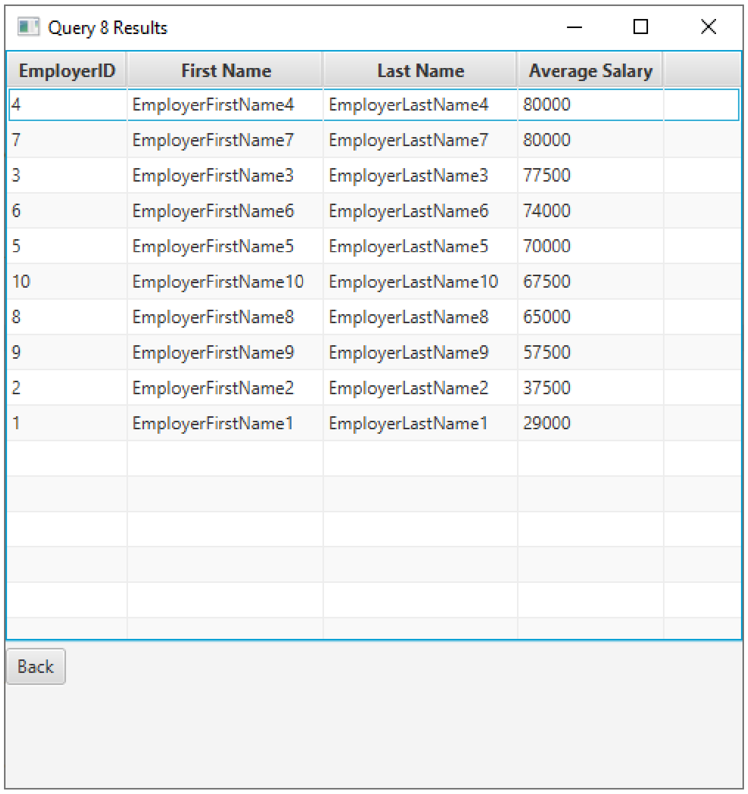
Figure 11: Query 6 Results



Query 7 Results

jobID	Company	Profession	Salary	Date Posted	Deadlin
1	Walmart	Cashier	30000	2023-09-23 00:00:00	2023-12-
2	Walmart	Greeter	28000	2023-09-24 00:00:00	2023-12-
3	Amazon	Warehouse Associate	35000	2023-09-25 00:00:00	2023-12-
4	Amazon	Delivery Driver	40000	2023-09-26 00:00:00	2023-12-
5	Google	Software Engineer	80000	2023-09-27 00:00:00	2023-12-
6	Google	Data Scientist	75000	2023-09-28 00:00:00	2023-12-
7	Microsoft	Software Developer	75000	2023-09-29 00:00:00	2023-12-
8	Microsoft	Project Manager	85000	2023-09-30 00:00:00	2023-12-
9	Apple	iOS Developer	80000	2023-09-30 00:00:00	2023-12-
10	Apple	Graphic Designer	60000	2023-09-25 00:00:00	2023-12-
11	Facebook	Social Media Manager	70000	2023-09-25 00:00:00	2023-12-
12	Facebook	Product Designer	78000	2023-09-25 00:00:00	2023-12-
13	Tesla	Mechanical Engineer	90000	2023-09-25 00:00:00	2023-12-
14	Tesla	Quality Assurance	70000	2023-09-25 00:00:00	2023-12-
15	Netflix	Content Writer	60000	2023-09-25 00:00:00	2023-12-

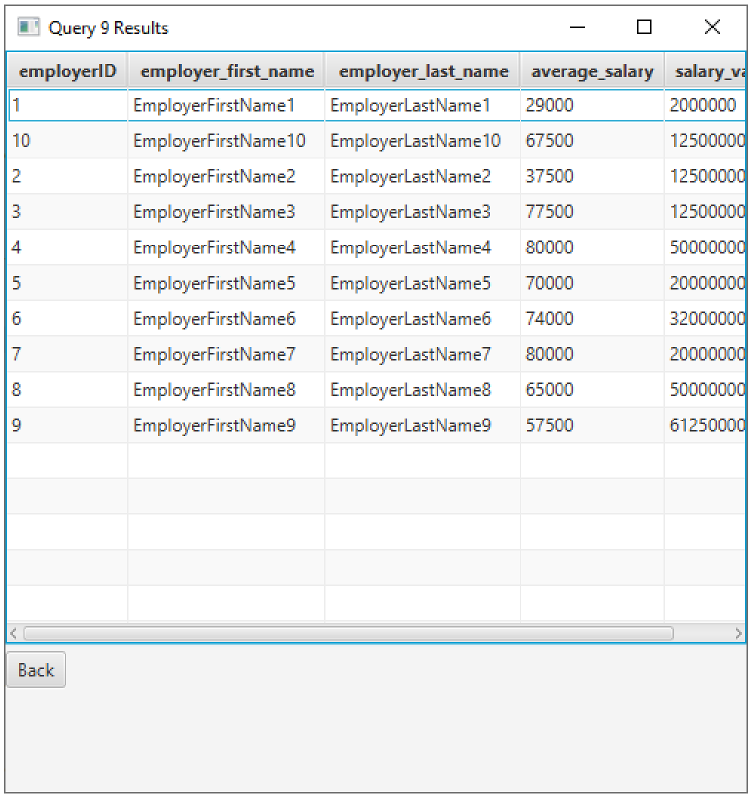
Figure 12: Query 7 Results



Query 8 Results

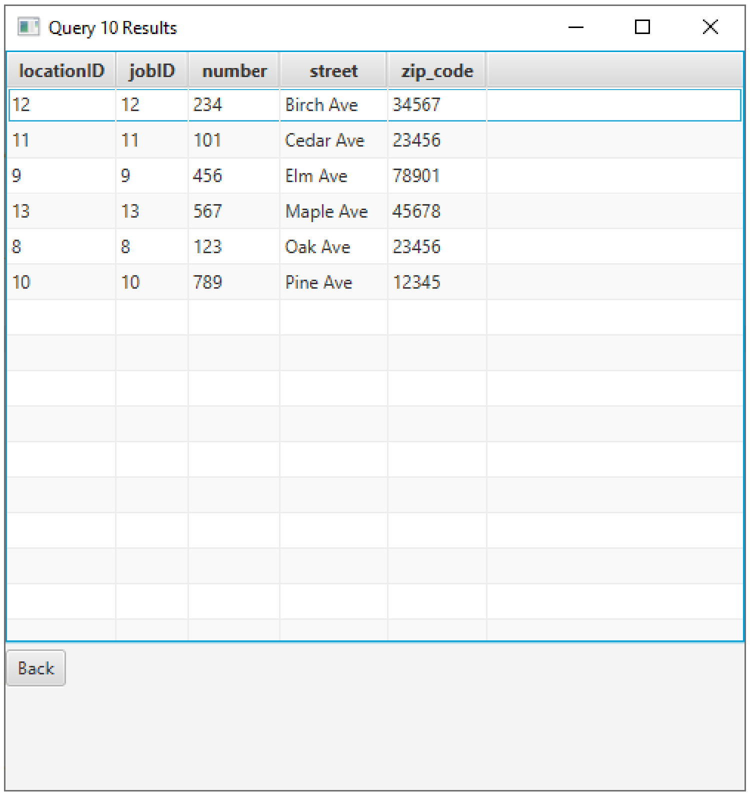
EmployerID	First Name	Last Name	Average Salary
4	EmployerFirstName4	EmployerLastName4	80000
7	EmployerFirstName7	EmployerLastName7	80000
3	EmployerFirstName3	EmployerLastName3	77500
6	EmployerFirstName6	EmployerLastName6	74000
5	EmployerFirstName5	EmployerLastName5	70000
10	EmployerFirstName10	EmployerLastName10	67500
8	EmployerFirstName8	EmployerLastName8	65000
9	EmployerFirstName9	EmployerLastName9	57500
2	EmployerFirstName2	EmployerLastName2	37500
1	EmployerFirstName1	EmployerLastName1	29000

Figure 13: Query 8 Results


Query 9 Results

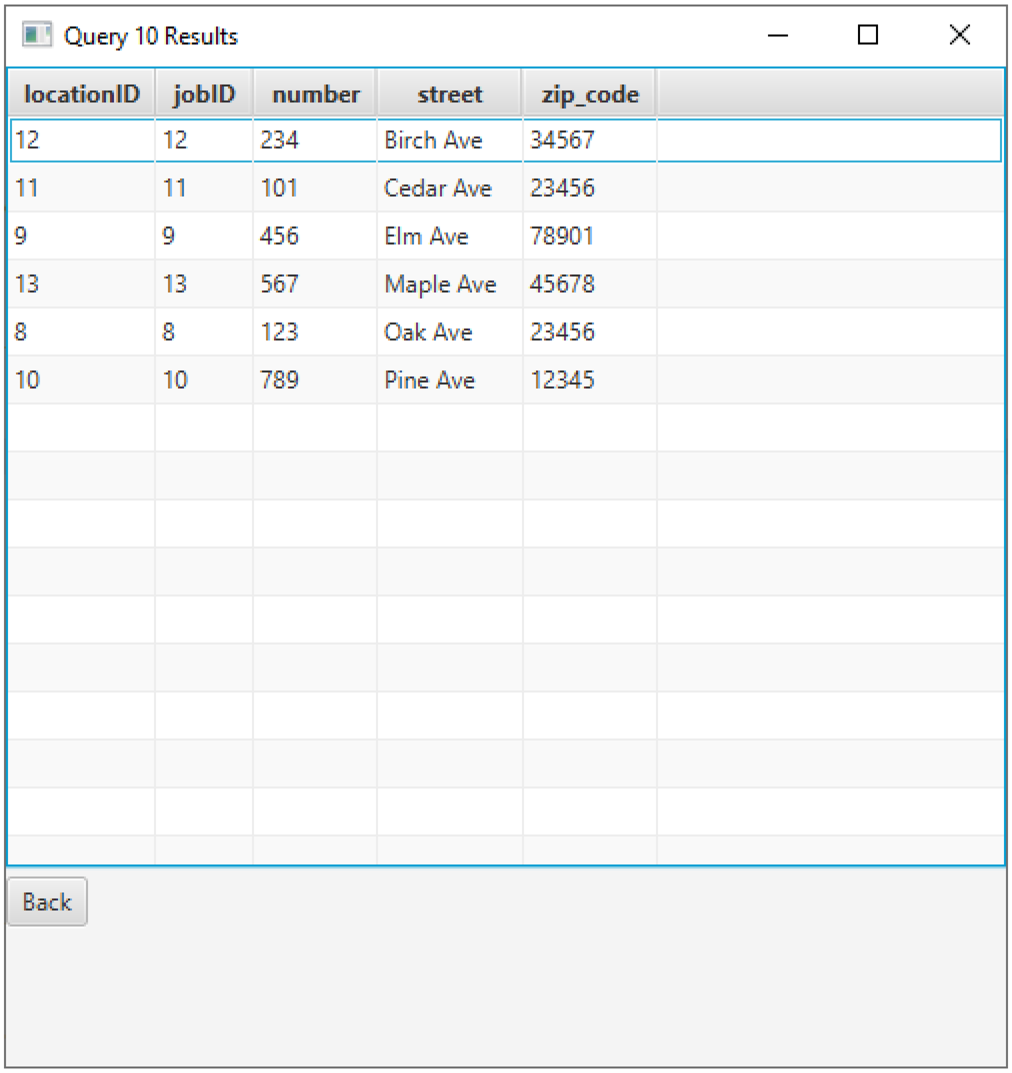
employerID	employer_first_name	employer_last_name	average_salary	salary_va
1	EmployerFirstName1	EmployerLastName1	29000	2000000
10	EmployerFirstName10	EmployerLastName10	67500	12500000
2	EmployerFirstName2	EmployerLastName2	37500	12500000
3	EmployerFirstName3	EmployerLastName3	77500	12500000
4	EmployerFirstName4	EmployerLastName4	80000	50000000
5	EmployerFirstName5	EmployerLastName5	70000	20000000
6	EmployerFirstName6	EmployerLastName6	74000	32000000
7	EmployerFirstName7	EmployerLastName7	80000	20000000
8	EmployerFirstName8	EmployerLastName8	65000	50000000
9	EmployerFirstName9	EmployerLastName9	57500	61250000

Figure 14: Query 9 Results


Query 10 Results

locationID	jobID	number	street	zip_code
12	12	234	Birch Ave	34567
11	11	101	Cedar Ave	23456
9	9	456	Elm Ave	78901
13	13	567	Maple Ave	45678
8	8	123	Oak Ave	23456
10	10	789	Pine Ave	12345

Figure 15: Query 10 Results



locationID	jobID	number	street	zip_code
12	12	234	Birch Ave	34567
11	11	101	Cedar Ave	23456
9	9	456	Elm Ave	78901
13	13	567	Maple Ave	45678
8	8	123	Oak Ave	23456
10	10	789	Pine Ave	12345

Figure 16: Query 11 Results

Closing Remarks

In the course of this database design project, we have traversed through multiple phases, from the initial logical design to the implementation and documentation phases. Each stage brought its unique set of challenges, and we sought effective solutions to ensure the successful development of our database application.

Application Description and Requirements:
We began by finalizing the application's scope and functionalities. This phase played a pivotal role in setting the foundation for our subsequent design decisions. Clear communication with the TA helped us define the information we expected from the application.

Entity-Relationship Model (ER Model):
The ER model served as a visual representation of the database structure, capturing the relationships and dependencies between entities. It facilitated a comprehensive understanding of the data requirements and paved the way for schema design.

Schema Design and Database Construction:
Translating the ER model into actual database tables using Oracle marked the transition from conceptual design to practical implementation. The creation and population of tables, along with the execution of simple queries, laid the groundwork for the subsequent phases.

Designing Views and Simple Queries:
The design of views and the formulation of simple queries demonstrated our ability to extract meaningful information from the database. This step was crucial in evaluating the effectiveness of our initial schema design and refining it based on real-world querying needs.

Advanced Queries and Unix Shell Implementation:
The implementation of advanced queries, including joins, set operations, and statistical functions, showcased the versatility of our database application. The use of Unix shell scripts provided a text-based interface, offering a glimpse into the functionality that would later be developed into a graphical user interface.

Normalization and Functional Dependencies:
The normalization process, spanning 3NF and BCNF, aimed to ensure data integrity and eliminate redundancies. Understanding and verifying functional dependencies allowed us to make informed decisions about the design's normalization level.

Database Application Development with UI:
The transition from shell scripts to a Java-based UI marked a significant milestone. The incorporation of a user-friendly interface enhances the accessibility and usability of our application. Additionally, the inclusion of dummy data and advanced reports adds depth to our project.

Documentation and Relational Algebra Notation:
Documentation played a crucial role in capturing the evolution of our project. The inclusion of Relational Algebra (RA) notation in our SQL queries enhances the clarity of our database design and query expressions.

In conclusion, this project provided a holistic learning experience, integrating theoretical concepts with hands-on implementation. The iterative nature of the design process allowed us to adapt and refine our approach, ultimately leading to a robust and efficient database application.