

INFS3200 Advanced Database Systems

Prac 3: Data Linkage (5%)

*Semester 2, 2022***Due:** Week 11 – Friday October 14th at 4pm**Submission:** via Blackboard → [Assessment](#) > [Practicals](#) > [Practical Three](#) > [Submission](#)

Introduction

Learning objectives

- Learn how to use *JDBC/cx_Oracle* to interact with *Oracle DBMS*, including the operations to *create* a table, *update* a table, *query* a table, and so on.
- Learn how to measure the performance characteristics of data linkage.
- Understand various string similarity measures, e.g., edit distance and Jaccard coefficient, for field-level data linkage.

Marking Scheme:

- **2 marks:** Receive one mark for completing each of the two Task 1 questions.
- **1 mark:** Complete Task 2; each of the two Task 2 questions is worth 0.5 marks.
- **1 mark:** Complete Task 3.
- **1 mark:** Complete Task 4.

Submission Format

Screenshot your results for each task and compile them into a document with appropriate explanation. Keep your explanations terse but make sure you contain all necessary information. Make sure your screenshots contain **your student ID** (since your student ID will be included in the name of the users you created) as proof of originality. Export your report into a **PDF document**. Copy your scripts/code into a subdirectory named `scripts` or `src` and pack your scripts and report into a zip file. Your zip file should be named appropriately, eg `JohnnyExample_s1234567890_prac1.zip`. Please format your document and code nicely to assist the tutor's marking process. A poorly formatted document may receive a reduced mark. **Due 4pm on Friday the 14th of October, 2022.**

Late Penalties (from the ECP)

“Where an assessment item is submitted after the deadline, without an approved extension, a late penalty will apply. The late penalty shall be 10% of the maximum possible mark for the assessment item will be deducted per calendar day (or part thereof), up to a maximum of seven (7) days. After seven days, no marks will be awarded for the item. A day is considered to be a 24 hour block from the assessment item due time. Negative marks will not be awarded.”

Part 1: Preparation of the Restaurant Data Set

In this part, our objective is to prepare data for subsequent tasks. There are **two options** for data storage: (1) import data into the Oracle database, or (2) store data directly, on-disk, as CSV files. Both solutions are implemented and available in **either Java or Python**. Although both options are available, we recommend you try the database option, as you can learn more about how to interact with the database using Java/Python code and libraries. But, ultimately, the choice is yours.

Option 1: Import data to Oracle database through JDBC/cx Oracle → *Scroll down for Option 2*

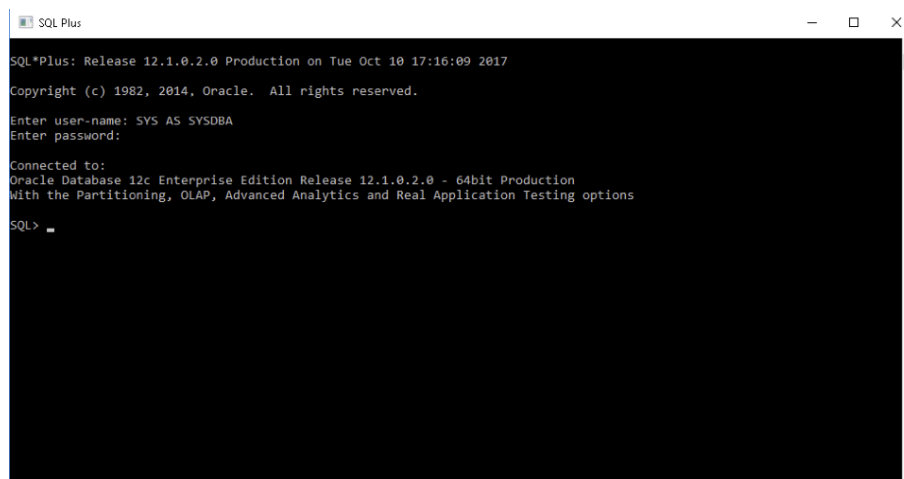
This option requires the following software:

- (1) Oracle platform used in Prac1 & Prac2,
- (2) Java JDK/Python library (Java 8/Python 3.8 recommended), and
- (3) Java/Python IDE (or a text editor if you are brave :-)

Here we give an example of Java with *Eclipse*, but others, like *IntelliJ IDEA* for Java, *PyCharm* for Python, or *Vim* will work the same. You need to go through the following 4 steps.

Step 1: Log in and Create Users

In this part, we first use “*SQL Plus*” to create a database user, and then we need to connect the user to “*SQL Developer*” and interact with the Oracle database. In SQL Plus Command Line window, login to Oracle with a default username “*SYS AS SYSDBA*” and password “*Password1*”, as shown below. (Recall: the password might be “*Password1!*” depending on your configuration).



```
SQL*Plus: Release 12.1.0.2.0 Production on Tue Oct 10 17:16:09 2017
Copyright (c) 1982, 2014, Oracle. All rights reserved.
Enter user-name: SYS AS SYSDBA
Enter password:
Connected to:
Oracle Database 12c Enterprise Edition Release 12.1.0.2.0 - 64bit Production
With the Partitioning, OLAP, Advanced Analytics and Real Application Testing options
SQL> _
```

Follow the commands below to create a user: **Please change “S1234567” to your own ID.**

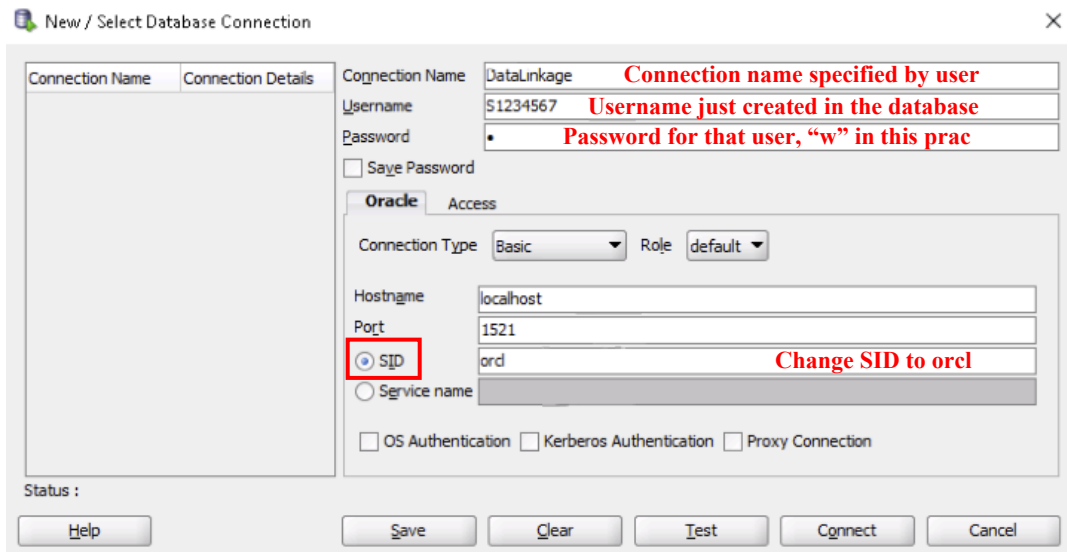
```
/*Enable user creation*/
ALTER SESSION SET "_ORACLE_SCRIPT"=TRUE;

/* Create a user named "S1234567" (student id) with password "w" */
CREATE USER S1234567 IDENTIFIED BY w ACCOUNT UNLOCK DEFAULT
TABLESPACE "USERS" TEMPORARY TABLESPACE "TEMP" PROFILE "DEFAULT";

/* Grant DBA privilege to "S1234567" */
GRANT DBA TO S1234567;
```

Step 2: Use Oracle SQL Developer

Open SQL Developer and connect the user that we have just created.

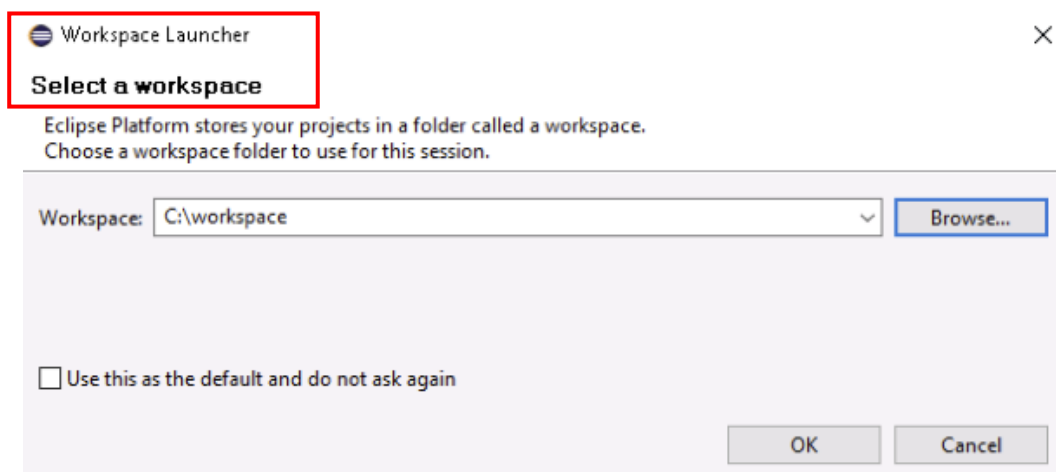


Click the **green “+”** button to connect to the database. Fill the connection information in the prompted dialog window shown as above. The “**connection name**” is specified by the user, and the “**username**” is the “*S1234567*” we have just created. “**Password**” is the password for that user, i.e., “*w*” in this case. You also need to change **SID** to “**orcl**”. Then press the “**Connect**” button to connect to the user “*S1234567*”.

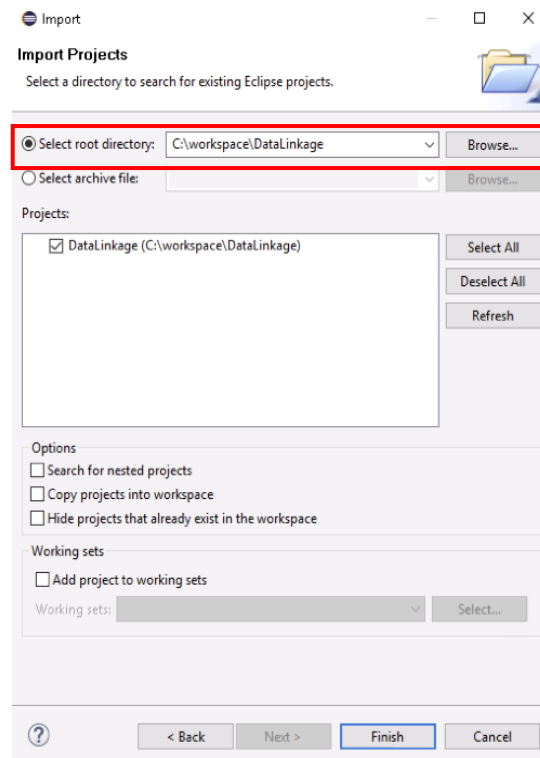
Step 3: Import the Data Linkage Project via Java IDE

In this step, we import the Java code template to your *Java IDE*. Here we use *Eclipse* as our example. Other IDEs work in a similar way, and you don’t need to use Eclipse if you know what you are doing with a different IDE. From the **Windows 10 “Start”** menu, you can search for “*Eclipse*”.

When you open the “*eclipse*” software, a dialog window will be prompted asking you to select a workspace, as shown below. Choose a folder where you want your *Eclipse* project to be stored.



Extract the *DataLinkage* project (downloaded from Blackboard course website as [P3.zip](#)) to the workspace. At the eclipse “*Project Explorer*” panel, mouse right-click and then choose the “import” function. From the prompted dialog window, click “**General – Existing Projects into Workspace**” and then press the “**Next**” button. Browse the file folder to find the *DataLinkage* project in your workspace and press “**Finish**”, as shown below. Now the project has been successfully imported into your workspace.



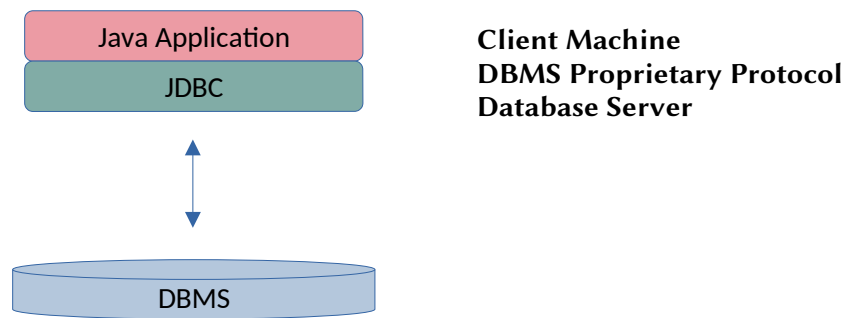
Step 4: Understand *JDBC/cx_Oracle* Connection to Oracle DBMS

The JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database (the counterpart in Python is [cx_Oracle](#), open a command prompt and enter “***python -m pip install cx_Oracle --upgrade***” to install *cx_Oracle* to Python). JDBC helps you to write Java applications that manage these three programming activities:

- Connect to a data source, like a database,
- Send queries and update statements to the database,
- Retrieve and process the results received from the database in answer to your query.

In order to connect a Java application with the Oracle database, typically you need to follow five steps to perform database connectivity. In the following example, we use a database supported by Oracle 12c. Hence, we need to know some related information for the Oracle database management:

- **Driver class:** The driver class for the Oracle database is “[oracle.jdbc.driver.OracleDriver](#)”.
- **URL:** The URL for Oracle 12c database is “[jdbc:oracle:thin:@localhost:1521:orcl](#)” where “*jdbc*” is the API, “*oracle*” is the database, “*thin*” is the driver, “*localhost*” is the server name on which Oracle is running (we may also use the IP address), “*1521*” is the port number, and “*orcl*” is the Oracle service name.
- **Username:** In this Prac, please use the “[S1234567](#)” user that you have just created.
- **Password:** Password is assigned at the time of creating a user, i.e., “**w**” in this case.



The following simple code fragment gives an example of a five-step process to connect to an Oracle database and perform certain queries.

Example 1:

In this code fragment, we create a connection *con* to the database and store the query result of “SELECT * FROM table” to the result set *rs* and display them. Browse the *DataLinkage* project in your IDE and find the package “*Oracle*”. We provide various Java classes for interacting with the Oracle database, including:

```

import java.sql.*;

class OracleCon {
    public static void main(String args[]) {
        try {
            // step1 load the driver class
            Class.forName("oracle.jdbc.driver.OracleDriver");

            // step2 create the connection object
            Connection con =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl", "S1234567",
"w");

            // step3 create the statement object
            Statement stmt = con.createStatement();

            // step4 execute query
            ResultSet rs = stmt.executeQuery("select * from table");

            // step5 run through the returned tuples
            while (rs.next())
                System.out.println(rs.getInt(1) + " " + rs.getString(2) + " " +
rs.getString(3));

            // step6 close the connection object
            con.close();

        } catch (Exception e) {
            System.out.println(e);
        }
    }
}

```

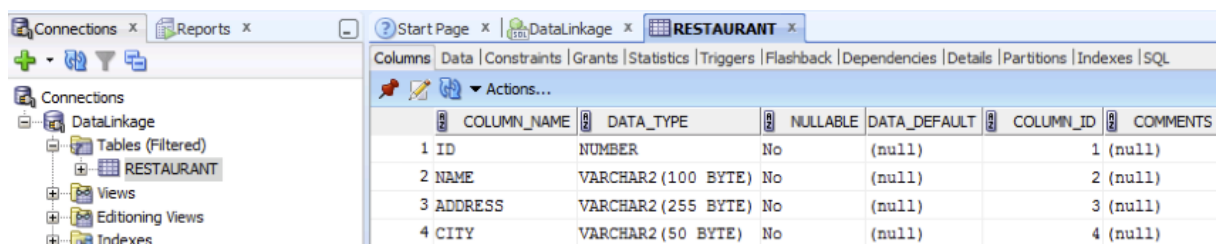
- *DBConnection*: Basic connection settings and functionalities.
- *CreateTable*: **Create** a table in the database
- *InsertTable*: **Insert** tuples into a table
- *DeleteTable*: **Delete** tuples from a table
- *ReadTable*: **Select** tuples from a table
- *DropTable*: **Drop** a table from the database
- In order to play with this functionality, you need to first change the username to the user you just created in the `DBConnection.java` class.

CREATE TABLE

The dataset we are going to use in this Prac is a list of *restaurants* at various locations. The table contains four attributes: ID, Name, Address, and City. We use the following SQL statement to create this table in the database:

```
CREATE TABLE RESTAURANT (ID NUMBER NOT NULL,  
NAME VARCHAR2(100 BYTE) NOT NULL,  
ADDRESS VARCHAR2(255 BYTE) NOT NULL,  
VARCHAR2(50 BYTE) NOT NULL,  
RESTAURANT_PK PRIMARY KEY(ID) ENABLE);
```

Run the CreateTable.java class to execute this SQL query using JDBC. Open CreateTable.java, right-click the class, and choose “**Run As – Java Application**”. When the program terminates, check the table you created in SQL Developer.

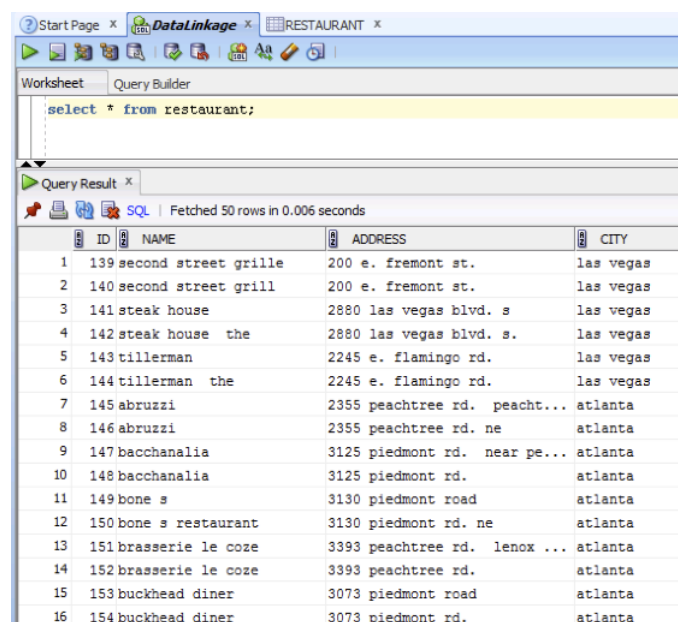


The screenshot shows the SQL Developer interface with the 'RESTAURANT' table selected. The 'Columns' tab is active, displaying the table's structure. The table has four columns: ID (NUMBER), NAME (VARCHAR2(100 BYTE)), ADDRESS (VARCHAR2(255 BYTE)), and CITY (VARCHAR2(50 BYTE)). All columns are NOT NULL and have a default value of (null). The table is part of a database connection named 'DataLinkage'.

COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1 ID	NUMBER	No	(null)	1	(null)
2 NAME	VARCHAR2(100 BYTE)	No	(null)	2	(null)
3 ADDRESS	VARCHAR2(255 BYTE)	No	(null)	3	(null)
4 CITY	VARCHAR2(50 BYTE)	No	(null)	4	(null)

INSERT INTO TABLE

The class InsertTable.java reads all the restaurant records from the excel file we have provided in this Prac, i.e. “data\restaurant.csv”, and inserts these records one by one into the RESTAURANT table using JDBC. Run InsertTable.java and then check the result in SQL Developer.

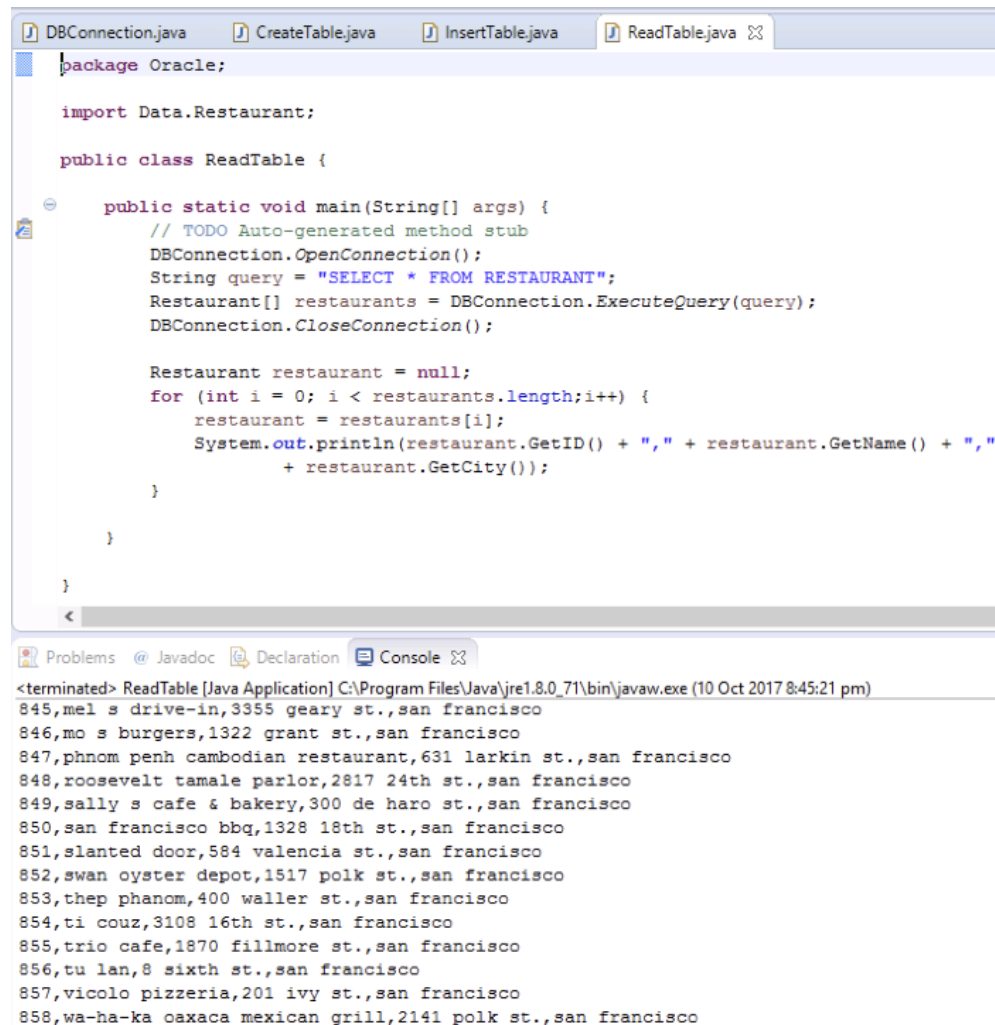


The screenshot shows the SQL Developer interface with the 'Query Result' tab active. The query 'select * from restaurant;' has been executed, and the results are displayed in a table. The table has four columns: ID, NAME, ADDRESS, and CITY. The results show 16 rows of data, including restaurant names, addresses, and cities.

ID	NAME	ADDRESS	CITY
1	139 second street grille	200 e. fremont st.	las vegas
2	140 second street grill	200 e. fremont st.	las vegas
3	141 steak house	2880 las vegas blvd. s	las vegas
4	142 steak house the	2880 las vegas blvd. s.	las vegas
5	143 tillerman	2245 e. flamingo rd.	las vegas
6	144 tillerman the	2245 e. flamingo rd.	las vegas
7	145 abruzzo	2355 peachtree rd. peacht...	atlanta
8	146 abruzzo	2355 peachtree rd. ne	atlanta
9	147 bacchanalia	3125 piedmont rd. near pe...	atlanta
10	148 bacchanalia	3125 piedmont rd.	atlanta
11	149 bone s	3130 piedmont road	atlanta
12	150 bone s restaurant	3130 piedmont rd. ne	atlanta
13	151 brasserie le coze	3393 peachtree rd. lenox ...	atlanta
14	152 brasserie le coze	3393 peachtree rd.	atlanta
15	153 buckhead diner	3073 piedmont road	atlanta
16	154 buckhead diner	3073 piedmont rd.	atlanta

SELECT FROM TABLE

The class `ReadTable.java` reads all the restaurant records from table `RESTAURANT` in the database by executing the SQL query, i.e., “**SELECT * FROM RESTAURANT**”, using JDBC. It then prints out these records (ID, Name, Address, City) on the *Eclipse* Console, as shown below. There should be **858** records in total, which can be seen at the end of the printed screen.



The screenshot shows the Eclipse IDE with the `ReadTable.java` file open. The code defines a `main` method that connects to a database, executes a `SELECT * FROM RESTAURANT` query, and prints the results. The console output shows the first 14 records of the restaurant table, each with an ID, name, address, and city.

```
package Oracle;

import Data.Restaurant;

public class ReadTable {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        DBConnection.OpenConnection();
        String query = "SELECT * FROM RESTAURANT";
        Restaurant[] restaurants = DBConnection.ExecuteQuery(query);
        DBConnection.CloseConnection();

        Restaurant restaurant = null;
        for (int i = 0; i < restaurants.length; i++) {
            restaurant = restaurants[i];
            System.out.println(restaurant.GetID() + "," + restaurant.GetName() + ","
                               + restaurant.GetCity());
        }
    }
}
```

<terminated> ReadTable [Java Application] C:\Program Files\Java\jre1.8.0_71\bin\javaw.exe (10 Oct 2017 8:45:21 pm)
845,mel s drive-in,3355 geary st.,san francisco
846,mo s burgers,1322 grant st.,san francisco
847,phnom penh cambodian restaurant,631 larkin st.,san francisco
848,roosevelt tamale parlor,2817 24th st.,san francisco
849,sally s cafe & bakery,300 de haro st.,san francisco
850,san francisco bbq,1328 18th st.,san francisco
851,slanted door,584 valencia st.,san francisco
852,swan oyster depot,1517 polk st.,san francisco
853,thep phanom,400 waller st.,san francisco
854,ti couz,3108 16th st.,san francisco
855,trio cafe,1870 fillmore st.,san francisco
856,tu lan,8 sixth st.,san francisco
857,vicolo pizzeria,201 ivy st.,san francisco
858,wa-ha-ka oaxaca mexican grill,2141 polk st.,san francisco

Option 2: Read data from the CSV File → Scroll up for Option 1

In this option, you can use either a Java IDE or Python IDE, determined by the language that you prefer to use. Again, you can also write your programs in a text editor and execute them in a command line environment if you prefer to do so.

Java IDE

In Java, we provide you with the data loader functionality in `CSVLoader.java` under the *Data* package. The function `restaurantLoader()` in `CSVLoader.java` reads the restaurant information from the CSV file. We call this function in three files, namely `NestedLoopByName.java`, `NestedLoopByNameED.java`, and `NestedLoopByNameJaccard.java`. Therefore, you should enter these three files respectively and enable the CSV reader by uncommenting the following line:

```
Restaurant[] restaurants = CSVLoader.restaurantLoader("data\\restaurant.csv");
```

Make sure you comment out the **option 1** part in each file:

```
// option1: read data from database
// DBConnection.OpenConnection();
// String query = "SELECT * FROM RESTAURANT";
// Restaurant[] restaurants = DBConnection.ExecuteQuery(query);
// DBConnection.CloseConnection();

// option2: read data from csv file, switch to this option by commenting the above code and uncommenting next line
Restaurant[] restaurants = CSVLoader.restaurantLoader( filePath: "data\\restaurant.csv");
```

Be sure to perform the same process to all three files.

Python IDE

In Python, we provide you with the data loader functionality in `csv_loader.py`. Since we also provide two data loading option in Python, we choose the data loading from CSV as default in `nested_loop_by_name.py`, `nested_loop_by_name_ed.py` and `nested_loop_by_name_jaccard.py`. You can switch between two options in the following code snippet.

```
16 # option1: read data from database
17 '''
18 con = db.create_connection()
19 cur = db.create_cursor(con)
20 string_query = "SELECT * FROM RESTAURANT"
21 cur.execute(string_query)
22 restaurants = []
23 for rid, name, address, city in cur:
24     restaurant = res()
25     restaurant.set_id(rid)
26     restaurant.set_name(name)
27     restaurant.set_address(address)
28     restaurant.set_city(city)
29     restaurants.append(restaurant)
30
31 cur.close()
32 con.close()
33 '''
34
35 # option2: read data from csv file, switch to this option by commenting the above code and uncommenting next line
36 restaurants = csv.csv_loader()
```


Assessment Task 1

Read the code in `NestedLoopByName.java/nested_loop_by_name.py` and focus on the data loading part. Understand how data are loaded into `restaurants` array and complete the following data statistics tasks **in the given class**

`DataStatistics.java/data_statistics.py`:

- Count the number of restaurant records whose city is “la” and “los angeles”, respectively.
- Count total number of distinct values in `city` attribute (**Hint**: use `HashSet` in Java or `set` in Python).

There are two ways of completing this task:

- (1) Load data from Oracle database or CSV file to “`Restaurant[] restaurants`” object using the method implemented in the `NestedLoopByName.java` class or in the `nested_loop_by_name.py` script, then obtain corresponding results by processing the “`restaurants`”, or
- (2) Write SQL queries that retrieve the task results directly from database; send these queries through `JDBC/cx_Oracle` (like the example shown in Example 1) and print the result to screen.

Hint: in Java, if you want to complete this task through SQL queries, you may also create an alternative/new `ExecuteQuery(query)` method in `DBConnection.java`, as this specific method was designed for parsing restaurant records to a `Restaurant[]` array.

Please screenshot both your code in `DataStatistics.java/data_statistics.py` and the running results. They should be included within one image. The format of the results should be similar to the following example (the values will be different):

La: 1000
los angeles: 100000
Number of Distinct Values in City: 1234

Part 2: Measure the Performance of Data Linkage

There is some duplication in the original *restaurant* dataset, and we will use *Data Linkage* techniques to detect these duplications, i.e., pairs of records that refer to the same real-world entity. For example, the following two records actually represent the same restaurant:

- 5, “hotel bel-air”, “701 stone canyon rd.”, “bel air”
- 6, “bel-air hotel”, “701 stone canyon rd.”, “bel air”

Nested-Loop Join for Data Linkage

The nested-loop join, a.k.a **nested iteration**, uses one join input as the outer input table and the other one as the inner input table. The outer loop consumes the outer input table row by row. The inner loop, executed for each outer row, searches for matching rows in the inner input table. The pseudo-code below shows its workflow.

```
For each tuple  $r$  in  $R$  do
  For each tuple  $s$  in  $S$  do
    If  $r$  and  $s$  satisfy the join condition
      Output the tuple  $\langle r, s \rangle$ 
```

In the simplest case, the search scans an entire table or index, which is called a naive nested loop join. In this case, the algorithm runs in $O(|R|*|S|)$ time, where $|R|$ and $|S|$ are the number of tuples contained in tables R and S respectively and can easily be generalized to join any number of relations. Furthermore, the search can exploit an index as well, which is called an indexed nested loop join. A nested loop join is particularly effective if the outer input is small and the inner input is pre-indexed and large. In many small transactions, such as those affecting only a small set of rows, index nested loop joins are superior to both merge joins and hash joins. In large queries, however, nested loop joins are often not the optimal choice.

In this Prac, we adopt a nested-loop method to self-join the `RESTAURANT` table for data linkage. We first consider *perfect matching* on the **Name** attribute. In other words, we link two restaurant records (i.e., they refer to the same entity) only if their names are identical:

- 1, “arnie morton's of chicago”, “435 s. la cienega blv.”, “los angeles”
- 2, “arnie morton's of chicago”, “435 s. la cienega blvd.”, “los angeles”

The programs named `NestedLoopByName.java` or `nested_loop_by_name.py` provide implementations of this basic join algorithm. It first reads all restaurant records from the Oracle database using JDBC, and then self-joins the table via a nested-loop join. If two records have the same **Name** value, the algorithm outputs this linking pair, i.e., **id1_id2** where **id1** and **id2** are the **IDs** of these records respectively.

Precision, Recall, and *F*-measure

We can regard the problem of data linkage as a *classification* task. Specifically, for each pair of records r and s , we predict a binary class label: “0” or “1”, where “1” means we believe these two records refer to the same entity and hence can be linked. Naturally, a data linkage algorithm is perfect if and only if:

- (1) **Precision**: all the linked pairs it predicts are correct, and
- (2) **Recall**: all possible linked pairs are discovered.

We provide a file `restaurant_pair.csv` which stores the *gold-standard* linking results, i.e., all the restaurant record pairs that refer to the same real-world entity. Suppose that D represents the set of linked pairs obtained by the data linkage algorithm, and D^* is the set of gold-standard linking results. The algorithm is regarded as perfect if the two sets are identical, i.e., $D=D^*$.

Precision and **Recall** are well-known performance measures that can capture the above intuition. For classification tasks, the terms *true positives*, *true negatives*, *false positives*, and *false negatives* are considered to compare the results of the classifier under test with trusted external judgments (i.e., **gold-standard**). The terms *positive* and *negative* refer to the classifier's prediction (sometimes known as the expectation), and the terms *true* and *false* refer to whether that prediction corresponds to the external judgment (sometimes known as the observation), as shown below. Given a linked pair `id1_id2` in this Prac, we can define each as follows:

- **True positive**, if it belongs to both D and D^*
- **False positive**, if it only belongs to D
- **False negative**, if it only belongs to D^*
- **True negative**, if it belongs to neither D nor D^*

		True condition	
Total population		Condition positive	Condition negative
Predicted condition	Predicted condition positive	True positive	False positive, Type I error
	Predicted condition negative	False negative, Type II error	True negative

Based on these terms, precision and recall are defined as follows, where tp , fp , and fn represent true positive, false positive, and false negative, respectively.

$$Precision = \frac{tp}{tp+fp} \quad \quad \quad Recall = \frac{tp}{tp+fn}$$

Often, there is an inverse relationship between precision and recall, where it is possible to increase one at the cost of reducing the other. For example, brain surgery can be used as an illustrative example of the trade-off. Consider a brain surgeon tasked with removing a cancerous tumour from a patient's brain. The surgeon needs to remove all of the tumour cells since any remaining cancer cells will regenerate the tumour. Conversely, the surgeon must not remove healthy brain cells since that would leave the patient with impaired brain function. The surgeon may be more liberal in the area of the brain they remove to ensure they have extracted all of the cancer cells. This decision *increases recall* but *reduces precision*. On the other hand, the surgeon may be more conservative in the amount of brain matter they remove to ensure they extract only cancer cells. This decision *increases precision* but *reduces recall*. Thus, greater recall increases the chances of removing healthy cells (a negative outcome) and increases the chances of removing all cancer cells (a positive outcome). Greater precision decreases the chances of removing healthy cells (a positive outcome) but also decreases the chances of removing all cancer cells (a negative outcome).

Therefore, another performance measure is used to consider the both precision and recall together, namely, *F-measure*, in which precision and recall are combined by their harmonic mean, as shown below.

$$F = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

Assessment Task 2

The `Measurement.java` and `measurement.py` classes implement the above measures, namely *precision*, *recall*, and *F-measure*. Please read and understand the “*CalcuMeasure*” function in `Measurement.java`. Explain the following concepts and explain which variable in the code are they correspond to (*count*, *result.size()*, *benchmark.size()*) in our data linkage problem.

Note that some concepts may require additional calculation on the existing variables. In addition, some may not derivable from the variables provided; if this is the case, just explain its meaning. **Include your explanation in the submitted document:**

- What are the true positive, true negative, false positive and false negative rates?
- What are the meanings of precision and recall in this case?

Hint: An example solution might be: “False positives are the records that appear in both the linkage result and the gold-standard, which is corresponding to *count/result.size()*. There are 120 false positive pairs.” or “The false positives are the records that appear in both the linkage result and the gold-standard; they are not calculated in the code.”

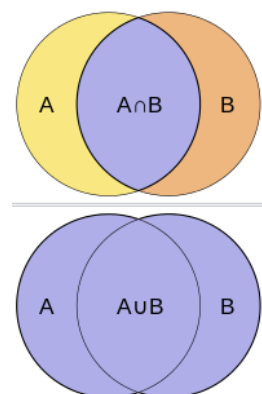
Part 3: Similarity Measures for Field-Level Data Linkage

In Part 2, we linked two restaurant records only if their names are identical. However, in practice, data is typically quite noisy, full of abbreviations, spelling mistakes, various entity representations, etc. Therefore, a better solution would be to calculate the similarity between records, which is a key issue in data linkage. We consider two string similarity measures in Task 3, i.e., *Jaccard coefficient* and *edit distance*, to estimate the similarity between restaurant names, so as to link corresponding restaurant records. You will see later how these similarity measures affect the performance of data linkage.

Jaccard Coefficient

Jaccard coefficient, also known as the Jaccard index or Intersection over Union, is a statistic method used for comparing the similarity and diversity of sample sets. The Jaccard coefficient measures similarity between finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{(|A| + |B| - |A \cap B|)}$$



Here, *A* and *B* are two sample sets, and $J(A, B)=1$ if *A* and *B* are both empty. As you can see, the Jaccard coefficient compares members for two sets to see which members are shared and which are distinct. It measures the similarity between two sets, with a range from 0% to 100%. The higher the percentage, the more similar the two sets. Consider the following simple example, how similar are these two sets?

- $A = \{0,1,2,5,6\}$
- $B = \{0,2,3,4,5,7,9\}$

→ $|A| = 5$, $|B| = 7$, $|A \cup B| = 9$, and $|A \cap B| = 3$, so the Jaccard index is 0.33.

In order to measure the similarity between two strings using Jaccard coefficient, the strings need to be converted into sets firstly. In this Prac, we consider a ***q*-gram** representation of each string. A *q*-gram is a contiguous sequence of *q* items from a given string. The items can be phonemes, syllables, characters, or words according to the application. A *q*-gram of size 1 is referred to as a “unigram”; size 2 is a “bigram”; size 3 is a “trigram” (or **3-gram**). Larger sizes are sometimes referred to by the value of *q* in modern language, e.g., “four-gram”, “five-gram”, and so on. Consider the string “University_of_Queensland”; we can transform it as a set of 3-grams with *each character* as the item: “University_of_Queensland” → {“##U”, “#Un”, “Uni”, “niv”, “ive”, “ver”, “ers”, “rsi”, “sit”, “ity”, “ty_”, “y_o”, “_of”, “of_”, “f_Q”, “_Qu”, “Que”, “uee”, “een”, “ens”, “nsl”, “sla”, “lan”, “and”, “nd#”, “d##”}, where using character “#” is optional to pad the beginning and ending of the string to get the complete set of 3-grams.

The Jaccard coefficient similarity measure based on *q*-grams has been implemented in the class `Similarity.java/similarity.py`.

Assessment Task 3

In `NestedLoopByNameJaccard.java/nested_loop_by_name_jaccard.py`, we link two restaurant records if the Jaccard coefficient of corresponding restaurant names exceeds a predefined threshold.

Run `NestedLoopByNameJaccard.java/nested_loop_by_name_jaccard.py` with different settings of “*q*” and “*threshold*” to see how these two parameters affect the output of the similarity measure, and therefore affect the performance of data linkage in terms of the output size and measurement result.

Your task is to test the influence of *each parameter* by altering its value to **five** different settings of your choice, (but make sure your values are valid) as well as fixing the other parameter. Do this process to **both** parameters and screenshot all of your *precision*, *recall* and *F-measure* results. Explain why the *precision/recall* increases/decreases based on your understanding. Include both your screenshots and your results in your submitted document. **Note** that, *q*=0 means we divide the original string into a *bag-of-words*.

Hint: You can build your results into a simple table like the following one.

Parameters	Precision	Recall	F1
q=2, threshold = 0.5	X	X	X
q=3, threshold = 0.5	X	X	X
...	X	X	X
...	X	X	X

Edit Distance

Given two strings A and B , their edit distance is the minimum number of edit operations required to transform A into B . Most commonly, the edit operations allowed are:

- **Insert** a character into a string;
- **Delete** a character from a string;
- **Replace** a character of a string by another character.

For these operations, edit distance is sometimes known as *Levenshtein* distance. For example, the edit distance between “cat” and “dog” is 3. Edit distance can be generalized to allow different weights for different kinds of edit operations. For instance, a higher weight may be placed on replacing the character “s” by the character “p”, than on replacing it by the character “a” (the latter being closer to “s” on the keyboard). Setting weights based on the likelihood of letters substituting each other is very effective in practice. However, to simplify the work in this Prac, we will only focus on the case in which **all edit operations have the same weight**.

The time complexity of computing the edit distance between two strings A and B is quadratic, that is, $O(|A| * |B|)$, where $|A|$ and $|B|$ denote the length of strings A and B respectively. The idea is to use a dynamic programming algorithm, where the characters in A and B are given in array form. The algorithm fills the (integer) entries in a matrix namely E , whose two dimensions equal to the lengths of the two strings whose edit distances is being computed. After execution, the $E[i][j]$ entry of the matrix will hold the edit distance between the strings consisting of the **first i** characters of A and the **first j** characters of B . There is a relation between $E[i][j]$ and $E[i-1][j-1]$, as one is a *sub-problem* of the other. **See the week 8 lecture if you are stuck**. Therefore, by using the dynamic programming algorithm, we can calculate the edit distance between two strings based on the edit distance of their *substrings*.

In particular,

- If “x” and “y” are identical, then $E[i][j] = E[i-1][j-1]$;
- If “x” and “y” are different, and we insert “y” for the first string, then $E[i][j] = E[i][j-1] + 1$;
- If “x” and “y” are different, and we delete “x” for the first string, then $E[i][j] = E[i-1][j] + 1$;
- If “x” and “y” are different, and we replace “x” with “y” for the first string, then $E[i][j] = E[i-1][j-1] + 1$;
- When “x” and “y” are different, $E[i][j]$ is the **minimum** of the three situations.

Some examples are available in the *Data Linkage Lecture Notes* and *Tutorial*. **It is worth noting that the edit distance is not a direct hint of the similarity between two strings**. Naturally, long strings could have more typing errors than short strings. In other words, we need to normalize the edit distance by string length in order to have a fair comparison of two strings. Since the maximum possible edit distance between any two strings A and B is $\max(|A|, |B|)$, we further calculate *edit similarity* as follows:

$$\text{Edit Similarity}(A, B) = 1 - \frac{(\text{Edit Distance}(A, B))}{(\max(|A|, |B|))}$$

Assessment Task 4

Please complete the implementation of **edit distance** in the `Similarity.java` or `similarity.py` classes, and then run either the `NestedLoopByNameED.java` or the `nested_loop_by_name_ed.py` program to see how **edit similarity** affects the performance of data linkage. As above in Task 3, you need to report the algorithm's **precision**, **recall**, and **F-measure** with **five** different settings of the “*threshold*” parameter, and provide a brief analysis of the trends.

Include both the screenshots of the results under different threshold settings and your explanation in your submitted document.

Note that edit distance is different from edit similarity. The edit distance similarity is implemented as `CalcuEDSim/calc_ed_sim` in Java/Python, which is a normalized similarity. The programs will automatically calculate **edit similarity** by calling your implementation of the **edit distance** function.

Hint: You can construct an example to test if your edit distance implementation works correctly. In Java, find edit distance and Jaccard coefficient using the code in `Similarity.java`:

1. Open `Similarity.java`.
2. Make a public main method to implement following lines as shown in the code below:

```
public static void main(String[] args) {  
  
    String str1 = "University";  
    String str2 = "Unvesty";  
    int out = Similarity.CalcuED(str1, str2);  
    System.out.println("Edit Distance = "+ out);  
    double out2 = Similarity.CalcuJaccard(str1, str2, 2);  
    System.out.println("Jaccard Coefficient = "+ out2)  
};
```

In Python, do the similar process in `similarity.py` as follows:

```
str1 = "University"  
str2 = "Unvesty"  
out = calc_ed(str1, str2)  
print("Edit Distance = ", out)  
out2 = calc_jaccard(str1, str2, 2)  
print("Jaccard Coefficient = ", out2)
```

Based on your understanding of these two measures, you can check if your code returns the expected value.