INFS3200 Advanced Database Systems

# Prac 3: Data Linkage (5%)

*Semester 1, 2022*

**Due time:**      4:00pm, Friday 20 May 2022 (Week 11)
**Submission:**   Submit your work online (INFS3200 Course Website)

## Introduction

**Learning objectives:**

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

**Marking Scheme:**

This Prac carries 5 marks for 5% of this course.

1. **2 marks:** Complete two missions in Task 1, each mission is accounted for 1 mark.
2. **1 mark:** Complete Task 2, the correct answer for each bullet point is worth 0.5 marks.
3. **1 mark:** Complete Task 3.
4. **1 mark:** Complete Task 4.

Please include your screenshots of the data import and data linkage results (*precision, recall, f-measure* and *time*) in a word/pdf document, as well as your answers to the task questions. It is not mandatary to include your student ID in every screenshots as we can check the originality through your code submission.

The submission should be a **zip/rar** file, which includes both the answer document and your source code files. Please format your document and code nicely to help tutor's marking process. A poorly formatted document may receive a reduced mark. *Submit your work to the Blackboard website of the course by 16:00 (i.e., 4pm), Friday 20 May 2022.*

**Late Penalties** (from 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 Oracle database, or (2) store data as CSV files. Both solutions are implemented and available in either Java or Python. Although both options are available, we recommend you to try the database option, as you can learn more about how to interact with database using Java/Python code.

# Option 1: Import data to Oracle database through *JDBC/cx_Oracle*
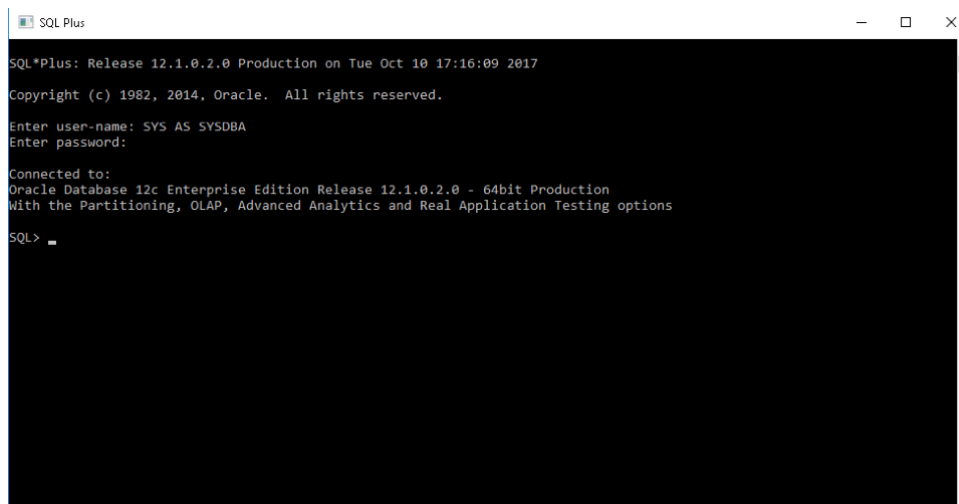
This option requires the following software:

        (1) Oracle platform used in Prac1 & Prac2,
        (2) Java JDK/Python library (Java 8/Python 3.8 recommended), and
        (3) Java/Python IDE.

Here we give an example of Java with *Eclipse*, but others, like *Intellij IDEA* for Java or *PyCharm* for Python, 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, then we need to connect user to "*SQL Developer*" and interact with an Oracle database. In SQL Plus Command Line window, login to Oracle with a default username "*SYS AS SYSDBA*" and password "*Password1!*", as shown below.



Follow the commands below to create a user:

```
/*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" */
```
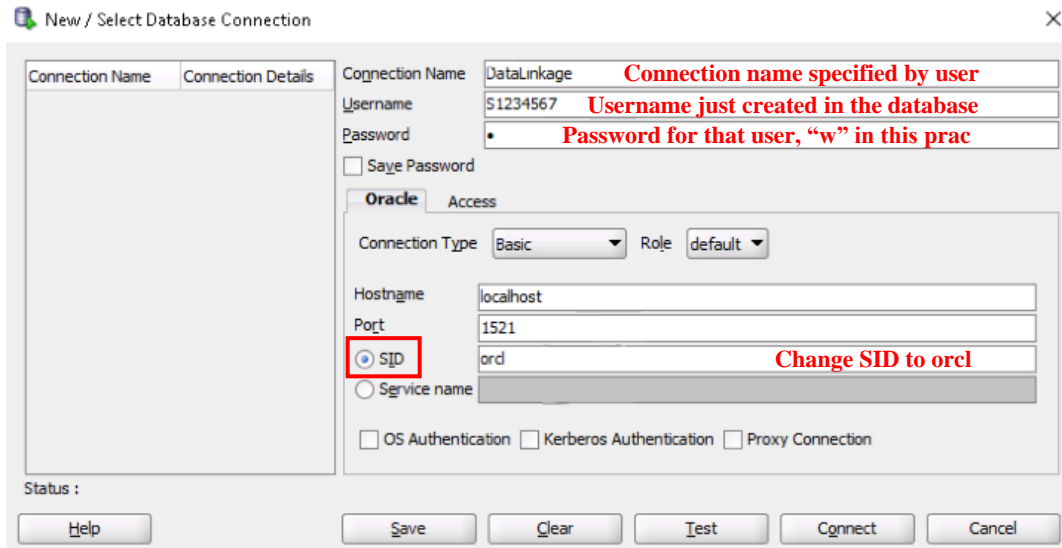
GRANT DBA TO S1234567;

Same as that of the previous Prac1 & Prac2, please change "S1234567" to your student ID.

## 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 "**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. From the **Windows 10** "**Start**" menu, you can search the "*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.
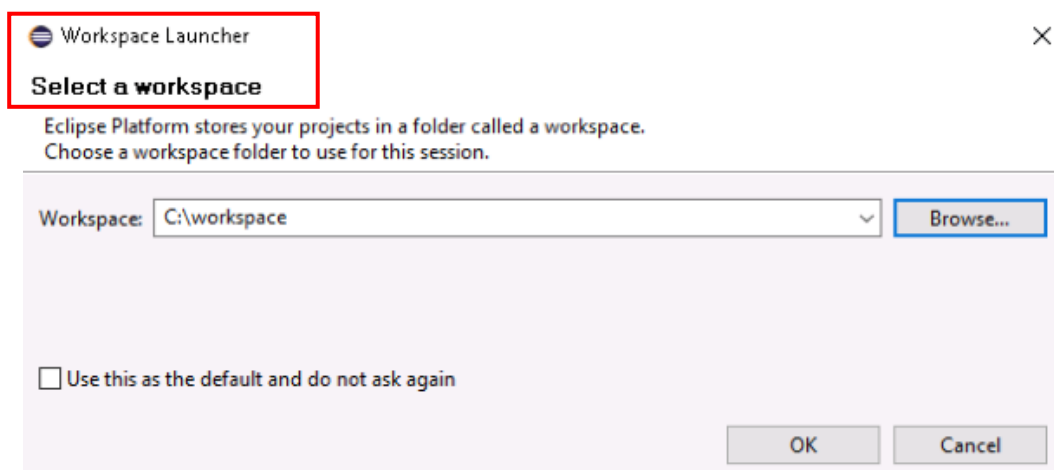
Extract the *DataLinkage* Java project downloaded from Blackboard course website to the workspace. At the eclipse "**Project Explorer**" panel, mouse right-click and then choose the "import" function. From the prompted dialog window, click "**General – Existing Projects into Workspace**" and then press 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.
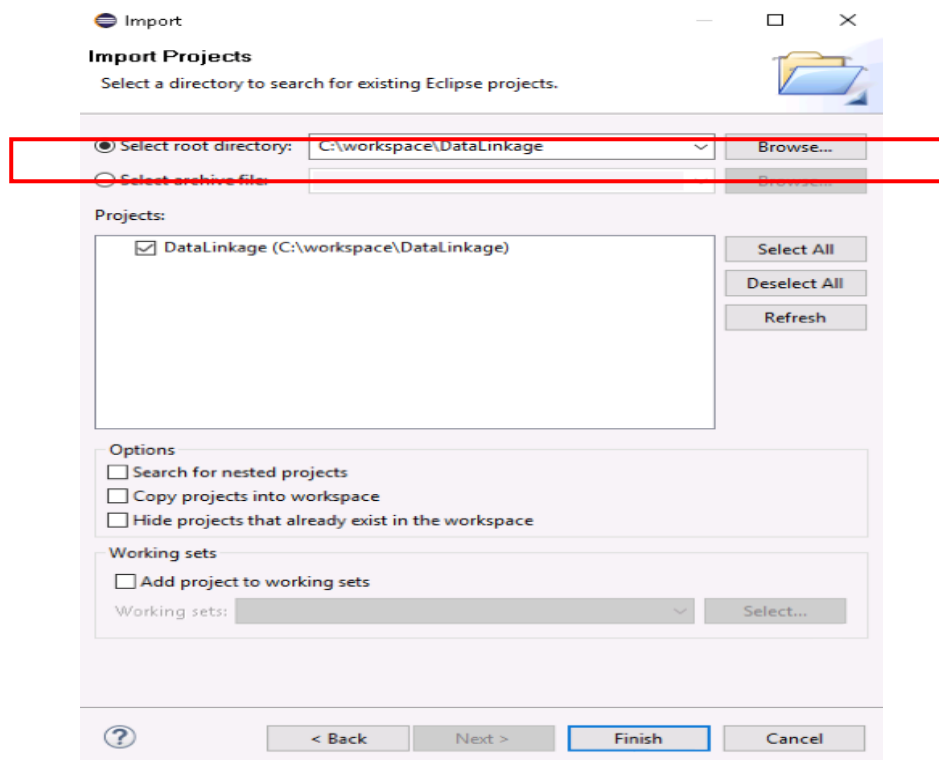


## Step 4: Understand *JDBC/cx_Oracle* Connection to Oracle DBMS

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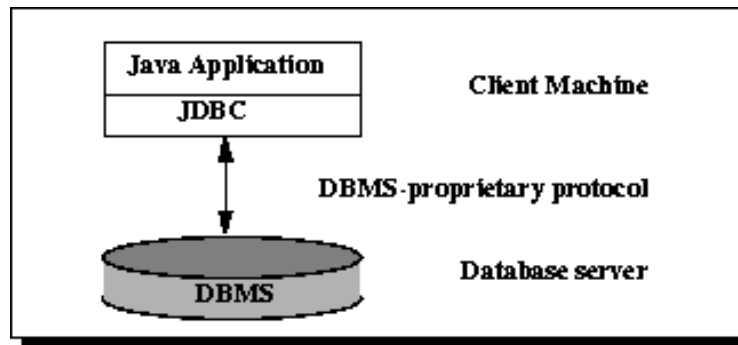
The following simple code fragment gives an example of a five-step process to connect to an Oracle database and perform certain queries.

**Example 1**:

```java
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");
            while (rs.next())
                System.out.println(rs.getInt(1) + "  " + rs.getString(2) + "  " + rs.getString(3));

            // step5 close the connection object
            con.close();

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

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:

- *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
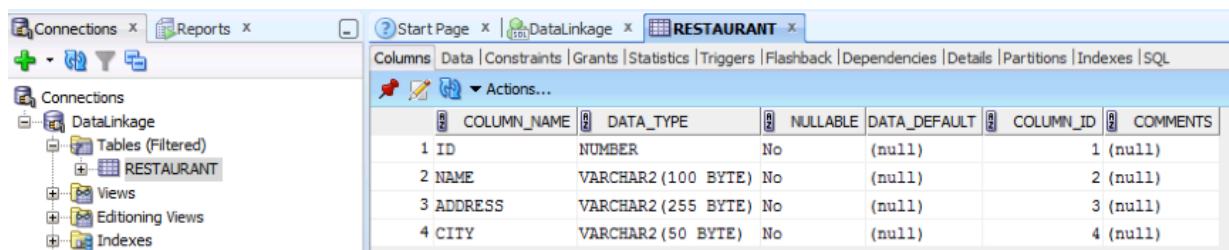- *DropTable*: **Drop** a table from the database

In order to play with these functionalities, you need to first change the username to the user you just created in *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.



## 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 insert these records one by one into the RESTAURANT table using JDBC. Run **InsertTable.java** and then check the result in SQL Developer.



6

## SELECT FROM TABLE

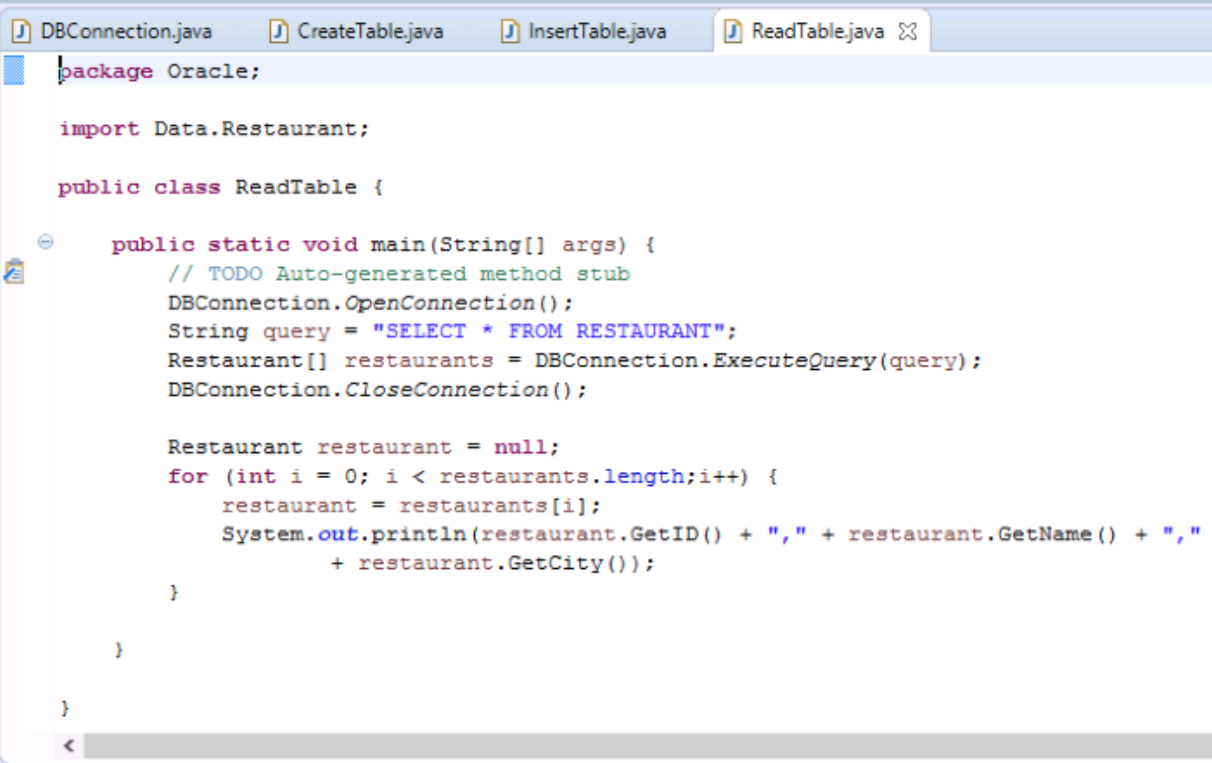The class **ReadTable.java** reads all the restaurant records from table RESTAURANT in the database by executing the SQL query, i.e., "**SELECT * FROM RESTAURANT**", using JDBC. It then prints out these records (**ID, Name, Address, City**) on the *Eclipse* Console, as shown below. There should be 858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

```java
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());
        }

    }

}
```

```
Problems  @ Javadoc  Declaration  Console
<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

In this option, IDE (integrated development environment) is used, which can be either a Java IDE or Python IDE, determined by the language that you prefer to use.

## 1. Java IDE:

In Java, we provide you with the data loader functionality in *CSVLoader.java* under *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");

In the meantime, comment the **option 1** part in each file, the result is shown as follows:

```
//       // 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.

## 2. 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()
```

***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 "new york" and "new york city", 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 *NestedLoopByName.java/ nested_loop_by_name.py*, then obtain corresponding results by processing the "**restaurants**", or

(2) Write SQL queries that retrieve the task results directly from database, send such query through *JDBC/cx_Oracle* (like the example shown in Example 1) and print the result to screen.

(**Note:** in Java, if you want to complete this task through SQL query, you may also change the *ExecuteQuery(query)* in *DBConnection.java,* as it is initially designed for parsing restaurant records to a *Restaurant[]* array. Writing a new parser for your new SQL is a better idea than changing *ExecuteQuery(query)* directly as it is called by other classes as well.)

*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 is similar as follows (The actual value is not the same as the example):

```
new york, 1000
new york city, 10000
Number of distinct values in city: 1024
```

# Part 2: Measure the Performance of Data Linkage

There are some duplications 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"

## 1. 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*
>       *Then output the tuple <r,s>*

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 index 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, e.g.,

- 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"

Program namely, *NestedLoopByName.java/nested_loop_by_name.py* is the implementation of this algorithm. It first reads all restaurant records from the Oracle database using JDBC, and then self-joins the table by 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 "**ID**"s of these records respectively.

## 2. 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 "**data\\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 intuitions. 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., <span style="color:red">**gold-standard**</span>). 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, it is like:

- **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\**

| Predicted condition | Total population | True condition | |
|---|---|---|---|
| | | Condition positive | Condition negative |
| | 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.

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

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 she removes to ensure she has extracted all the cancer cells. This decision increases recall but reduces precision. On the other hand, the surgeon may be more conservative in the brain she removes to ensure she extracts only cancer cells. This decision increases precision but reduces recall. It can be seen that, greater recall increases the chances of removing healthy cells (negative outcome) and increases the chances of removing all cancer cells (positive outcome). Greater precision decreases the chances of removing healthy cells (positive outcome) but also decreases the chances of removing all cancer cells (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.

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

***Task 2***: Program *Measurement.java/measurement.py* class implements above performance 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, and some may not derivable from the variables provided (then only explain its meaning). **Include your explanation in the submitted document:**

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

An example answer could be as such format (the answer in the example is not correct): "The false positive is 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 positive is the records that appear in both the linkage result and the gold-standard, it is not calculated in the code."

# Part 3: Similarity Measures for Field-Level Data Linkage

In above Part 2, we link two restaurant records only if their names are identical. However, datasets, in practice, are always informal and noisy, full of abbreviations, spelling mistakes, various entity representations, etc. Therefore, a better solution would be to calculate the similarity between records, which is also the main issue of 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 to link corresponding restaurant records. You will see later how these similarity measures affect the performance of data linkage.

# 1. Jaccard Coefficient

Jaccard coefficient, also known as 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:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$
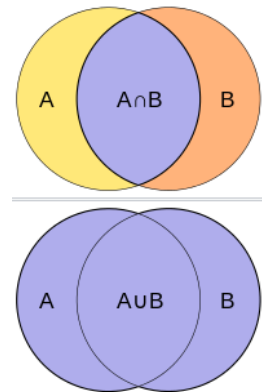
Here, A and B are two sample sets, and **J(A, B)=1** if A and B are both empty. As you can see, the Jaccard coefficient compares members for two sets to see which members are shared and which are distinct. It measures the similarity between two sets, with a range from 0% to 100%. The higher the percentage, the more similar the two sets. Consider the following simple example, how similar are these two sets? Obviously, J(A,B) = |A∩B| / |A∪B| = |{0,2,5}| / |{0,1,2,3,4,5,6,7,9}| = 3/9 = 0.33.

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

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 *q-gram* representation of a string. 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. As an example mentioned in Week-8 Lecture Notes and Tutorial 7, 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 for the completeness of 3-gram.

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

***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.

*Test the influence of each parameter by altering its value to five different settings (up to your choice, but make sure your values are valid) as well as fixing the other parameter. Do such process to both parameters and screenshot all 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.
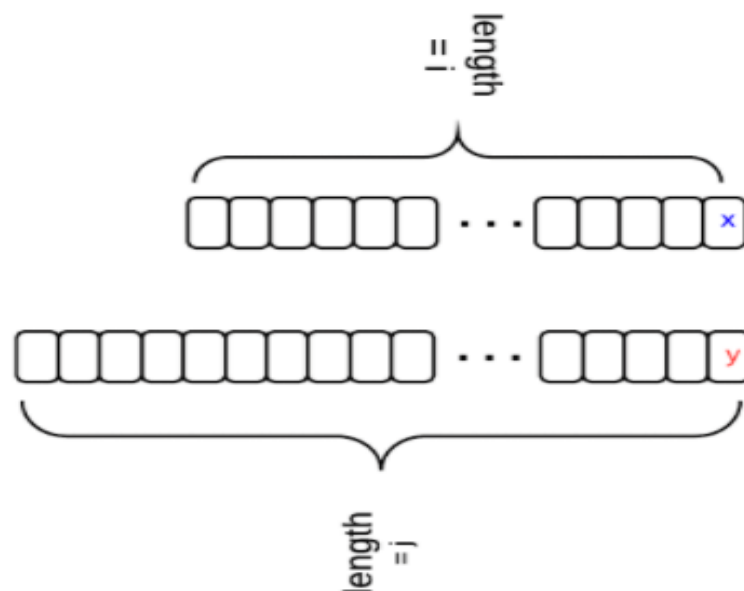
## 2. Edit Distance

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 for this purpose 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. In fact, edit distance can be generalized to allowing 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 in this way depending on the likelihood of letters substituting for 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.

It is well-known on time complexity of computing the edit distance between two strings is *O(|A|\*|B|)*, where |A| and |B| denote the length of strings A and B respectively. The idea is to use the 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 ED, whose two dimensions equal to the lengths of the two strings whose edit distances is being computed. The *ED[i][j]* entry of the matrix will hold (after the algorithm is executed) 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 *ED[i][j]* and *ED[i-1][j-1]*. Assume that we transform from one string to another. The first string has length *i* and its last character is "x", and the second string has length *j* and its last character is "y". The following diagram shows the relation.



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
  *ED[i][j] = ED[i-1][j-1]*;
- If "x" and "y" are different, and we insert "y" for the first string, then

13

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

| | | | f | | a | | s | | t | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | **0** | *1* | 1 | **2** | 2 | **3** | 3 | **4** | 4 |
| c | | 1 | *1* | 2 | 2 | 3 | 3 | 4 | 4 | 5 |
| | | 1 | 2 | *1* | 2 | 2 | 3 | 3 | 4 | 4 |
| a | | 2 | 2 | 2 | *1* | 3 | 3 | 4 | 4 | 5 |
| | | 2 | 3 | 2 | 3 | *1* | 2 | 2 | 3 | 3 |
| t | | 3 | 3 | 3 | 3 | 2 | 2 | 3 | 2 | 4 |
| | | 3 | 4 | 3 | 4 | 2 | 3 | 2 | 3 | 2 |
| s | | 4 | 4 | 4 | 4 | 3 | 2 | 3 | 3 | 3 |
| | | 4 | 5 | 4 | 5 | 3 | 4 | 2 | 3 | 3 |

Above is an example of using the dynamic programming algorithm to calculate the edit distance between strings "fast" and "cats". Some more examples are available in Week-8 Lecture Notes and in T7 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:

$$Sim(A, B) = 1 - \frac{ED(A, B)}{max(|A|, |B|)}$$

***Task 4***: Please complete the implementation of edit distance in *Similarity.java/similarity.py*, run *NestedLoopByNameED.java/nested_loop_by_name_ed.py* to see how edit distance affects the performance of data linkage. Same as the above Task 3, you need to report the algorithm's ***precision***, ***recall***, and ***f-measure*** with five different settings of the "*threshold*" and provide your understanding of the trends.

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

**Hint:** You can construct an example to test if our 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 screenshot 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:

```python
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.

**Note** that <u>edit distance</u> is different from <u>edit distance similarity</u>. The edit distance similarity is implemented as *CalcuEDSim/calc_ed_sim* in Java/Python, which is a normalized similarity.

<div align="center">

---ooo000OOO000ooo---

</div>