EECS 484 Projects | p2-fakebook-jdbc

Project 2: Fakebook JDBC

Worth	Released	Due
118 points (59 public, 59 private)	May 11th	**May 23rd at 11:55 PM ET**

Project 2 is due on **May 23rd at 11:55 PM EST**. Please refer to the EECS 484 SP23 Course Policies for more information on penalties for late submissions, late day tokens, and sick days.

Introduction

In Project 2, you will be building a Java application that executes SQL queries against a relational database and places the results in special data structures. We provide you the majority of the structure for the Java application and your job is to fill it with the query text and to process the results of the queries appropriately. This project will give you additional practice with standard SQL query practices in addition to hands-on experience with real-world database application programming.

Submissions

This project is to be done in teams of 2 students (recommended) or individually. Be sure to create your team on the Autograder.

Honor Code

By submitting this project, you are agreeing to abide by the Honor Code: "I have neither given nor received unauthorized aid on this assignment, nor have I concealed any violations of the Honor Code." You may not share answers with other students actively enrolled in the course outside of your teammate, nor may you consult with students who took the course in previous semesters. You are, however, allowed to discuss general approaches and class concepts with other students, and you are also permitted (and encouraged!) to post questions on Piazza.

The Public Data Set

The Fakebook data in Project 2 is structured similarly as the dataset you created for Project 1, except that there is neither a Messages nor Participants table. Use the `DESC` command to view the full schema of any of the public data tables. The tables have already been created and loaded with data. The full list of all the public tables can be found in PublicFakebookOracleConstants.java but is also provided here for your reference:

- Public_Users
- Public_Friends
- Public_Cities
- Public_User_Current_Cities
- Public_User_Hometown_Cities
- Public_Programs
- Public_Education
- Public_User_Events
- Public_Albums
- Public_Photos
- Public_Tags

Every row of two users *(user1_id, user2_id)* in *Public_Friends* will meet the invariant *user1_id* < *user2_id*. This enforces the constraint that users cannot be friends with themselves, and the structure of the table prevents friendships being listed more than once.

The tables are stored under the schema of `project2.` To access the tables in SQL*Plus, you should use `project2.<tableName>` . **You should use this access approach only when running your queries through SQL*Plus interactive mode, NOT in your Java implementation**. There is a separate access mechanism when implementing your queries in Java (see StudentFakebookOracle.java).

Starter Files

Download the starter files (p2-starter_files.tar.gz).

The compression (`p2-starter_files`) contains a directory named `project2` and you should not change its organization.

PublicFakebookOracleConstants.java

Do not modify this file. It defines the Oracle schema, prefix and a series of non-modifiable table name variables that you will use to implement your queries.

FakebookOracleUtilities.java

Do not modify this file. It defines a single utility class, `FacebookArrayList`, that will be used for storing lists of data structures built up from your query results. This utility class exists for the purpose of customizing printing output.

FakebookOracleDataStructures.java

Do not modify this file. It defines a series of custom data structures that allow you to report your query results. Please familiarize yourself with these various data structures so that you are comfortable creating new instances of them and invoking their various augmentation functions. Example usages of these data structures are shown in comments in StudentFakebookOracle.java.

FakebookOracle.java

Do not modify this file. It defines the abstract parent class from which your Java application will derive. This base class defines the 9 abstract functions that you will have to implement; these function declarations have already been repeated for you in StudentFakebookOracle.java. In addition, this base class defines a series of printing functions that are used to output the results of your queries.

StudentFakebookOracle.java

This is the file in which you should implement your SQL queries. It defines the derived query class that implements the abstract functions defined by the parent `FakebookOracle` class. Each of the 9 required queries has its own function, which is commented to briefly describe the goal of the query (a full description of the queries is at Queries). Additionally, each function skeleton contains a comment showing how to use the necessary data structures for the query (defined in FakebookOracleDataStructures.java). You are encouraged to follow the given structure and create additional Oracle statements and/or try-catch blocks when necessary.

The bottom of the `StudentFakebookOracle` class defines 11 constant variables that you should use to reference the public dataset tables (defined in PublicFakebookOracleConstants.java). Please familiarize yourself with these variables, but do not modify them. **Any time you wish to use the name of a table in your query, select the appropriate variable and insert the variable into your query string. DO NOT hard-code the table names into your queries under any circumstances: you will fail Autograder private tests if you do so.**

Do not modify the class constructor, which appears at the top of the class definition, and do not remove any of the @Override directives.

You are not permitted to use any additional Java libraries for this project. All of the necessary import statements have already been included in this file.

FakebookOracleMain.java

Near the top of the file, fill in your uniqname and Oracle password to enable a connection. If you forget to add your credentials, you will get the error `java.sql.SQLException: ORA-01017: invalid username/password; logon denied` .

```
1  private static String username = "uniqname"; // replace with your uniqname
2  private static String password = "password"; // replace with your Oracle password (defa
```

This is the application driver. It can be invoked from the command line (preferably by the Makefile) and takes command line arguments that define which query/queries to execute and whether to print the output or measure the runtime.

You will NOT submit this file, so you do not need to worry about staff members obtaining your password. However, if you wish to change your SQL*Plus password, refer to Tools.

It is never recommended to store a password in plaintext, so we recommend you not use a password you use anywhere else for your SQL account and instead make something easy to remember that you will never use again.

ojdbc6.jar

This is a JAR file needed to compile your application. This driver has been tested with JDKs 1.7 and 1.8; we cannot guarantee its compatibility with other JDK versions.

Makefile

A Makefile allows you to easily compile, run, and clean your code. You are responsible for ensuring that your application compiles and runs using the unmodified Makefile, which will be utilized on the Autograder. There is no guarantee that staff members will be able to assist you in customizing or troubleshooting the Makefile if you choose to add or modify make targets.

- To compile your Java application, navigate to your project root directory (where the Makefile is located) and run `make` or `make compile` . This will compile silently if there are no errors and will print any compilation problems to the command line. Fix any errors that you have so that your code perfectly compiles. We will not be able to grade your submission unless it compiles correctly.
- To run your queries and view the output, you have two options. If you want to run all your queries to compare your output to the provided solution output file, run `make query-all` . To

run a single query to view the output, run `make queryN` where N is the query number. You may redirect output for output diff.

- To run your queries and measure their runtime, you again have two options. If you want to time all your queries, run make `time-all`. To measure the runtime of a single query, run `make timeN` where N is the query number. Make sure that your code runs without any error before attempting to measure its runtime.
- To remove the files generated by compilation, run `make clean`.

PublicSolution.txt

This file contains the expected output of each of the 10 queries (one implemented for you, 9 you will implement) when your Java application is executed against the public dataset. The output of running `make query-all` on your application should match this file exactly; any deviation indicates an error with your code. You can separate the individual query results into their own files so that you can test the outputs of single queries; if you choose to do this, do not omit the trailing blank lines after the output of a query or the query result header.

PublicTime.txt

This file contains the average runtime of the instructor implementation when the ten queries are executed against the public dataset on the CAEN system. To collect your runtime, use the command `make time-all`. Be aware that the runtime for this project is extremely unstable due to the JDBC Oracle connection mechanism, and may change drastically when you switch between platforms. You do not need to perform strictly better than our runtime to pass all tests on the Autograder. A reasonable buffer time is allowed on the Autograder.

Queries

You will implement 9 SQL queries, although some of the queries may actually consist of multiple individual queries. They are listed below with detailed specifications as to what fields to return and in what order. You should put your queries into the appropriate Java function in StudentFakebookOracle.java. The results of the queries should be placed in the appropriate data structures (defined in FakebookOracleDataStructures.java) as demonstrated by the skeleton code comments.

It is your responsibility to ensure that your queries are correct irrespective of the dataset upon which the queries are executed. We have provided you with sample correct output based on the public dataset, but **we will also be testing your implementations against a private dataset** to which you are not given access. Points for each query will be split between performance on the public and private datasets.

Implementation Approach and Rules

- **We highly recommend writing your SQL queries in plaintext and executing them interactively through SQL*Plus before transplanting them into the Java application function.** Remember that in SQL*Plus, you should access the public data set with `project2.<tableName>` as explained in The Public Data Set. In Java, you should access the public data set with the variables in StudentFakebookOracle.java. **The SQL*Plus CLI will provide more helpful error messages than the JRE** (Java Runtime Environment), making it easier to debug your solutions.
- When you submit to the Autograder, don't submit partially-completed queries; this will drag down the time it takes to grade your solution. Make sure that your file compiles with just the necessary return statement in the incomplete query functions.
- **You are not permitted to create additional tables in your implementations.** You can, however, create additional views. If you do so, use `stmt.executeUpdate()` to create views rather than `stmt.executeQuery()`. You can access views by plaintext; you do not need to access them through variables like the public data set tables. Be sure to drop views before closing the statement at the end of the query.
- When testing your application, the creation and dropping of views can get complicated if you have syntax errors in queries in the same function. For example, if a `CREATE VIEW` is executed successfully, but your query throws an error before the corresponding `DROP VIEW` statement is executed, the view will still exist the next time you test your code. If you encounter a syntax error in a query where you have created a view, you should manually drop the view in the SQL*Plus CLI. You may use `SELECT view_name FROM user_views;` to list all the views in your schema.
- If you wish to add print statements to your functions for debugging purposes, you should use the Java equivalent of C++'s `cout`, which is `System.out.println`. This function takes a string parameter and prints it to the standard output stream. Similarly, you can print to the standard error stream with `System.err.println`. **Be sure to remove any such print statements from your file before submitting, as they will cause you to fail Autograder tests.**

Runtime Efficiency

All of the make targets have a built-in 90-second timeout; any queries that take longer than this will automatically terminate. Such queries will receive a 0 on the Autograder. Generally, queries that take too long will produce no output.

The efficiency tests are worth 2 points per query, public and private. Your queries will need to pass correctness tests in order to be evaluated for efficiency.

To pass the efficiency tests, offload most, but not all, of the work to the DBMS. For example, you should not be attempting to sort data in Java; rather, use an `ORDER BY` clause in your query. In most

cases, offloading work to the DBMS will be faster. However, when deciding between manually loop through a `Resultset` in Java and performing an expensive join and subquery in SQL, avoid the expensive subquery. Parameterizing your queries with Java variables will often be faster.

Ultimately, timing several different approaches to your queries will be the surest way to decide the best way to gather data. Check PublicTime.txt for the instructors' runtime averages.

Query 0: Birth Months (Provided: 0 points)

This query has been implemented for you as an example

Query 0 asks you to identify information about Fakebook users' birth months. You should determine in which month the most Fakebook users were born and in which month the fewest (but at least 1) Fakebook users were born. If there are ties, pick the month that occurs earliest in the calendar year. For each of those months, report the IDs, first names, and last names of the Fakebook users born in that month; sort the users in ascending order by ID. You should also report the total number of Fakebook users that have a birth month listed. You can safely assume that at least one Fakebook user has listed a birth month.

Query 1: First Names (12 points)

Public: 6 points • Private: 6 points

Query 1 asks you to identify information about Fakebook users' first names.

- We'd like to know the longest and shortest first names by length. If there are ties between multiple names, report all tied names in alphabetical order.
- We'd also like to know what first name(s) are the most common and how many users have that first name. If there are ties, report all tied names in alphabetical order.

Hint: You may consider using the `LENGTH()` operation in SQL. Remember that you are allowed to execute multiple SQL statements in one query.

Query 2: Lonely mUsers (12 points)

Public: 6 points • Private: 6 points

Query 2 asks you to identify all of the Fakebook users with no Fakebook friends. For each user without any friends, report their ID, first name, and last name. The users should be reported in ascending order by ID. If every Fakebook user has at least one Fakebook friend, you should return an empty FakebookArrayList.

Query 3: Users who Live Away from Home (12 points)

Public: 6 points • Private: 6 points

Query 3 asks you to identify all of the Fakebook users that no longer live in their hometown. For each such user, report their ID, first name, and last name. Results should be sorted in ascending order by the users' ID. If a user does not have a current city or a hometown listed, they should not be included in the results. If every Fakebook user still lives in his/her hometown, you should return an empty FakebookArrayList.

Query 4: Highly-Tagged Photos (14 points)

Public: 7 points • Private: 7 points

Query 4 asks you to identify the most highly-tagged photos. We will pass an integer argument num to the query function; you should return the top num photos with the most tagged users sorted in descending order by the number of tagged users (most tagged users first). If there are fewer than num photos with at least 1 tag, then you should return only those available photos. If more than one photo has the same number of tagged users, list the photo with the smaller ID first.

For each photo, you should report the photo's ID, the ID of the album containing the photo, the photo's Fakebook link, and the name of the album containing the photo. For each reported photo, you should list the ID, first name, and last name of the users tagged in that photo. Tagged users should be listed in ascending order by ID.

Query 5: Matchmaker (16 points)

Public: 8 points • Private: 8 points

Query 5 asks you to suggest possible unrealized Fakebook friendships. We will pass two integer arguments, `num` and `yearDiff` to the query function; you should return the top `num` pairs of two Fakebook users who meet each of the following conditions:

- The two users are the same gender
- The two users are tagged in at least one common photo
- The two users are not friends
- The difference in the two users' **birth years** is less than or equal to yearDiff

The pairs of users should be reported in (and cut-off based on) descending order by the number of photos in which the two users were tagged together. For each pair, report the IDs, first names, and last names of the two users; list the user with the smaller ID first. If multiple pairs of users that meet the criteria are tagged in the same number of photos, order the results in ascending order by the smaller user ID and then in ascending order by the larger user ID. If there are fewer than num pairs of users that meet the criteria, you should return only those pairs that are viable.

For each pair of users, you should also report the photos in which they were tagged together. The information you should report is the photo's ID, the photo's Fakebook link, the ID of the album containing the photo, and the name of the album containing the photo. List the photos in ascending order by photo ID.

Query 6: Suggest Friends (16 points)

Public: 8 points • Private: 8 points

Query 6 asks you to suggest possible unrealized Fakebook friendships in a different way. We will pass a single integer argument, `num`, to the query function; you should return the top `num` pairs of Fakebook users with the most mutual friends who are not friends themselves. If there are fewer than `num` pairs, then you should return only those available pairs.

A mutual friend is one such that A is friends with B and B is friends with C, in which case B is a mutual friend of A and C. The IDs, first names, and last names of the two users who share a mutual friend should be returned; list the user with the smaller ID first and larger ID second within the pair and rank the pairs in descending order by the number of mutual friends. In the event of a tie between pairs, list the pair with the *smaller first ID* before the pair with the *larger first ID*; if pairs are still tied, list the pair with the *smaller second ID* before the pair with the larger second ID.

For each pair of users you report, you should also list the IDs, first names, and last names of all their mutual friends. List the mutual friends in ascending order by ID.

Hint: Remember that the friends table contains one direction of user IDs for each friendship. Consider creating a *bidirectional* friendship view.

Query 7: Event-Heavy States (12 points)

Public: 6 points • Private: 6 points

Query 7 asks you to identify the states in which the most Fakebook events are held. If more than one state is tied for hosting the most Fakebook events, all states involved in the tie should be returned, listed in ascending order by state name. You also need to report how many events are held in those state(s). You can assume that there is at least 1 Fakebook event.

Query 8: Oldest and Youngest Friends (12 points)

Public: 6 points • Private: 6 points

Query 8 asks you to identify the oldest and youngest friend of a particular Fakebook user. We will pass a single integer argument, `userID`, to the query function; you should return the ID, first name, and last name of the oldest and youngest friend of the Fakebook user with that ID. Notice that you

should not type convert the date, month and year fields using `TO_DATE`; instead, order them just as they are (numbers). If two friends of the user passed as the argument are born on the exact same date, report the one with the *larger user ID*. You can assume that the user with the specified ID has at least 1 Fakebook friend.

Query 9: Potential Siblings (12 points)

Public: 6 points • Private: 6 points

Query 9 asks you to identify pairs of Fakebook users that might be siblings. Two users might be siblings if they meet each of the following criteria:

- The two users have the same last name
- The two users have the same hometown
- The two users are friends
- The difference in the two users' **birth years** is strictly less than 10 years

Each pair should be reported with the smaller user ID first and the larger user ID second. The smaller ID should be used to order pairs relative to one another (smaller smaller ID first); the larger ID should be used to break ties (smaller larger ID first).

Submitting

The only deliverable for Project 2 is StudentFakebookOracle.java. There are 59 points for the public test cases and 59 points for the private test cases.

This project in particular takes a lengthy amount of time to grade (up to 20 minutes). **Please do not make submissions to the Autograder right after your latest submission, before you receive the results.** Please be patient and only contact the staff with concerns if you have been waiting more than 30 minutes without seeing the results.

Each team will be allowed 3 submissions per day with feedback; any submissions made in excess of those 3 will be graded, but the results of those submissions will be hidden from the team. Your highest scoring submission will be used for grading, with ties favoring your latest submission.

Due to an Oracle JDBC connection stability issue, you may observe connection reset exceptions or timeouts for one or multiple test cases of your submissions. You may make a private Piazza post requesting a rerun if your test cases are affected by these issues. However, please make such requests sparingly so that the course staff doesn't get overloaded with rerun requests. Without relying on the Autograder, you should use the public output provided to check the correctness of your code on CAEN.

After the project is due and your private test results are released, you may also submit a regrade request for the staff to rerun those tests **only if** your highest scoring submission was affected by these errors. We will post instructions on Piazza for doing this when the project is due.

Appendix

Java Syntax You Should Know

This project has been designed in such a way that you do not need to know or understand significant aspects of the Java programming language to successfully complete it. Later sections of the appendix contain in-depth treatments of some Java-specific tools that will be of paramount importance in implementing your application. However, there are a handful of major syntactical differences between Java and C++ that you should be familiar with, as they may impact your programming.

- Java has two types of objects: *primitives* (which are analogous to C++ built-in data types) and *references* (which are analogous to C++ pointers). There are no pointers in Java, but all references are dynamically-allocated with the `new` keyword. For example, to create a variable `obj` of type `Foo`, you would write `Foo obj = new Foo();`. You do not need to use `new` to create primitive variables.
- Java has garbage collection, so you do not need to manually deallocate memory even if you allocate references with the `new` keyword.
- Java has primitive and reference versions of all simple data types. The primitive version is first-letter lower-case (i.e. `long`) and the reference version is first-letter upper-case (i.e. `Long`). This has an important bearing on equality comparisons. We highly recommend you use the primitive version wherever necessary, as it is more idiomatic and more akin to C++.
- Java has two ways to compare equality. To compare equality of primitives or to determine if two references are the exact same object (analogous to checking if two pointers are the same in C++), use the standard `==` comparator. To test if two references are logically equivalent (as defined by the class's particular implementation), you must use the `.equals()` member function.
- Java strings are references and are declared with a capital letter `String`. To create a new string, you can write `String str = "ABC"` without the `new` keyword.
- Java has automatic string conversion built into every primitive and every object, so the standard string concatenation `operator+` can be used to combine strings, numeric types, and objects; an example of this is `String str = "abc" + 2 + "def";` in which the numeric value 2 is converted to a string and properly concatenated.

Statements and Result Sets

The primary JDBC tools you will be using to execute your queries are `Statements` and `ResultSets`. The appropriate Java libraries have already been imported for you, so you can simply use these tools to perform your queries.

`Statements` are JDBC objects against which you can execute queries and updates. Each of the query function skeletons has already created a `Statement` object named `stmt` that you can use without any additional hassle. However, if you ever want to create a new `Statement` object, you can copy the body of the try-with-resource statement, changing the name of the variable as necessary. To execute a query against a `Statement`, you should use the `Statement::executeQuery(String)` member function, which returns a `ResultSet`. To create or drop a view, you should use the `Statement::executeUpdate(String)` function, which has no return type. See the implementation of Query 0 for examples of how to execute queries.

`ResultSets` are essentially lists of rows that are returned by queries executed against a `Statement`. To loop over the list of results, you can use the `ResultSet::next()` function, which returns false when you have advanced past the last result. You can also use the `ResultSet::isFirst()` and `ResultSet::isLast()` functions to determine if a particular row is the first/last row in the query result, respectively. To extract a value of a column from the current row of a `ResultSet`, use either `ResultSet::getLong(arg)` or `ResultSet::getString(arg)`. The argument to these functions can either be the case-sensitive name of the column whose data you wish to extract or the 1-based column index of the column whose data you wish to extract. See the implementation of Query 0 for examples of how to navigate `ResultSets` and extract data from them.

You will be responsible for closing all of the resources you utilize in this project; specifically, you must close all your `Statements` and `ResultSets` using the `close()` member function. You should always do this last, when you no longer need the object, as doing so otherwise will make it impossible to complete your implementation. An important thing to note is that **when you reuse a `Statement` to execute another query, any `ResultSets` generated previously from that `Statement` will get automatically closed.** As such, the following Java snippet (with actual query strings omitted for brevity) will induce a runtime error:

```
1  Statement stmt = new Statement ( ... ) ;
2  ResultSet rst = stmt.executeQuery ( ... ) ;
3  while (rst.next ()) {
4      ResultSet rst2 = stmt.executeQuery ( ... ) ;
5      long val = rst.getLong (1);
6  }
```

The reason is that the reuse of `stmt` to generate the results stored in `rst2` causes `rst` to close. The attempt to access the data in `rst` will thus throw an exception. **If you want to use multiple**

`ResultSets` in this fashion, you must create a second `Statement` to use for the inner query. Make sure to create this statement outside of the loop, however, so that it doesn't get garbage collected and reinitialized every time through.

Additionally, closing a `Statement` will close any `ResultSets` generated by that `Statement`; however, we explicitly suggest that you separately close your `ResultSets` before you close your `Statements` for the surest resource management. See the implementation of Query 0 for an example of how to close your resources.

The ROWNUM Pseudocolumn

In the course of implementing the queries, you may find that you wish to select only the first N rows of results. Oracle SQL provides `ROWNUM`, a pseudocolumn that facilitates this desire, but it can be incredibly fidgety.

The way `ROWNUM` works is quite simple, but not always intuitive. After the `FROM` clause is evaluated (meaning all JOINs are completed), each row is assigned a monotonically increasing integer starting at 1; this value is stored in a pseudocolumn called `ROWNUM`, which allows us to access the value later on. Note, however, that these values are applied before the `WHERE` clause is evaluated and, more importantly, before the `ORDER BY` clause is evaluated. Because the order in which SQL returns results in the absence of an `ORDER BY` clause is undefined, the order in which the values are applied to the rows is likewise undefined.

Consider the following query, which should find the users named "Bob" with the 10 smallest IDs:

```
1  SELECT user_ID
2  FROM Users
3  WHERE first_name = 'Bob' AND ROWNUM <= 10
4  ORDER BY user_id;
```

Although this query looks exactly right, it will almost certainly not behave as we'd like. The reason is that the rows are numbered before they are sorted, and as mentioned, that numbering is applied in no particular order. So when the `WHERE` clause is evaluated, the only rows that are returned are those for users whose first name is "Bob" that happened to be in the first 10 rows to which rows were assigned. Not only may this query not return the 10 smallest ID'd users whose first name is "Bob," but it might not even return any results despite there being results to return.

Instead, we would have to write out query like this:

```
1  SELECT user_id
2  FROM
3      (SELECT user_id
4      FROM Users
```

```
    WHERE first_name = 'Bob'
    ORDER BY user_id)
WHERE ROWNUM <= 10;
```

Now, the inner query returns all the users whose first name is “Bob” and sorts them in the order we want. Row numbers are then assigned correctly to the ordered results. We can then filter using the `ROWNUM` pseudocolumn and retrieve the first 10 rows of the queries result set.

Note that the `ROWNUM` pseudocolumn does not error out if there aren’t “enough” rows to return. It is also possible to complete this project and earn full credit without using the `ROWNUM` pseudocolumn.

Acknowledgements

This project was written and revised over the years by EECS 484 staff at the University of Michigan. The most recent version was updated and moved to Primer Spec by Owen Pang.