EECS 484 W19 Project #2: Fakebook Oracle JDBC

EECS 484 W19 Staff

Due: February 22nd, 2019 at 11:59pm EST

In Project #2, you will be building a Java application that executes complex SQL queries against a relational database and places the results in special data structures for later analysis. We will provide you with a mechanism for accessing the data, as well as the majority of the structure for the Java application; your job will be to fill in the application skeleton 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 (an imitation of) real-world database application programming.

This project is to be done in teams of 2 students or individually. We recommend that you work in pairs. Students may participate in the same teams as in Project #1, or they may switch partners; students do not need any special permission to switch partners. Both members of each team will receive the same score; as such, it is not necessary for each team member to submit the assignment. A tool for finding teammates has been made available on Piazza. To create a team, follow these steps:

1. One team member creates the team by clicking the "Create Team' button on the autograder page
2. Follow the directions on the "Create Team" page to invite the second team member to join the team, typing his/her **uniqname** in the input box
3. The second team member will receive an email with instructions on how to use the eight-letter join code to join the team
4. **Do not make any submissions until the second member joins the team, otherwise they will be prevented from joining!**

Project #2 is due on **Friday, February 22nd** at **11:59pm EST**. If you do not turn in your project by that deadline, or if you are unhappy with your work, you may continue to submit up until Tuesday, February 26*th* at 11:59m EST (4 days after the regular deadline). Please refer to the course syllabus for more information on late days.

The University of Michigan College of Engineering Honor Code strictly applies to this assignment, and we will be thoroughly checking to ensure that all submissions adhere to the Honor Code guidelines. Students whose submissions are found to be in violation of the Honor Code will be reported directly to the Honor Council. You may not share answers with other students actively enrolled in the course, 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 on which Project #2 relies is structured in the same way as the data set you **created** for Project #1, except that there is neither a Messages table nor a Participants table. The table names (with an appropriate prefix, see below), column names, and data types are similar, though there might be minor changes. **Use the DESC command to view the full schema of any of the public data tables.**

We believe it important to point out the that every row of two users (*A, B*) in the Friends table will meet the invariant $A < B$. 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 account of one of the EECS 484 staff members, jiaqni. The prefix for the public data set is PUBLIC (not case-sensitive), so to access the tables, you should use jiaqni.PUBLIC_[tableName]. You should only use this access approach when running your queries through sqlplus interactive mode; there is a separate access mechanism when implementing your queries in Java (see the next section).

Starter Code

There is a zip file named starter_code.zip available for download on Canvas. The compression contains nine starter files. Most of the files are not to be modified in any manner whatsoever; specific permissions and directives regarding modification of the starter files are listed in the following detailed treatment of the starter files.

PublicFakebookOracleConstants.java

PublicFakebookOracleConstants.java defines a series of non-modifiable variables that you will use to implement your queries. You should not modify this file in any way whatsoever; doing so will cause your application to encounter errors during execution.

FakebookOracleUtilities.java

FakebookOracleUtilities.java defines a single utility class, FakebookArrayList, 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. As such, you should not modify this file in any way whatsoever, as your query output is highly reliant on the specific implementation of this utility class.

FakebookOracleDataStructures.java

FakebookOracleDataStructures.java defines a series of custom data structures that you will need to use to report the results of your queries. Please familiarize yourself with these various data structures so that you are comfortable creating new instances and invoking the structures' various augmentation functions. Example usages of these data structures are shown in comments in StudentFakebookOracle.java as described below. You should not modify this file in any way whatsoever, lest your application output be different that what is expected.

FakebookOracle.java

FakebookOracle.java defines the abstract parent class from which your Java application will derive. This base class defines the nine abstract functions that you will have to implement; these function declarations have already been repeated for you in StudentFakebookOracle.java (see below). In addition, this base class defines a series of printing functions that are used to output the results of your queries. You should not modify this file in any way whatsoever, or your code will likely fail to compile when run on the Autograder.

StudentFakebookOracle.java

StudentFakebookOracle.java defines the derived query class that implements the abstract functions defined by the parent FakebookOracle class. **This is the file in which you should be implementing your SQL queries.** You should not modify the existing skeleton code other than to remove comments and, obviously, add your personal query implementations; this means that any data structure definitions, return statements, and try-catch blocks should be maintained in full. Each of the nine required queries has its own function, which is commented to briefly describe what the goal of the query is (a full description of the queries can be found later in the specification); additionally, each function skeleton contains a comment showing how to use the necessary data structures for the query.

The bottom of the StudentFakebookOracle class defines a series of eleven class-private constant variables that you should use to reference the public data set tables; these variables are essentially redefinitions of those constants contained in PublicFakebookOracleConstants.java that have been placed in this class for ease of reference. Please familiarize yourself with these variables – but do not modify them – so that you can comfortably use them in your queries. 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, under any circumstances, hard-code the names of the tables into your queries: you will fail Autograder tests if you do so.**

Do not modify the class constructor, which appears at the top of the class definition. Neither should you remove any of the @Override directives.

FakebookOracleMain.java

FakebookOracleMain.java 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 should not change this file *except* the snippet below, where you should put your uniqname and current SQL*PLUS password.

```
private static final String username = ‘ ‘USERNAME HERE’ ’ ;
private static final String password = ‘ ‘PASSWORD HERE’ ’ ;
```

You will not be submitting 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, you are more than welcome to; the Project #1 spec describes how to do this.

Makefile

Makefile is a makefile that we have written you that allows you to easily compile, run, and clean your code. See “Running and Testing Your Application” below for details on how to use the Makefile. You may modify this file if you wish, but 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 add or modify make targets. **ojdbc6.jar**

ojdbc6.jar 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.

PublicSolution.txt

PublicSolution.txt contains the expected output of each of the ten queries (nine you will implement, one implemented for you; see below) when your Java application is executed against the public data set. 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 if you wish 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.

Queries

Overview

You will be implementing nine SQL queries, although some of the queries may actually consist of multiple individual queries that have to be separately executed and somehow combined to produce a final result. These queries 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, from which it will be invoked by FakebookOracleMain.java with the appropriate command line parameters. The results of the queries should be placed in the appropriate data structures as specified by the skeleton code comments.

It is your responsibility to ensure that your queries are semantically correct; that is, your queries should be correct irrespective of the data set upon which the queries are executed. If your query returns the correct solution in some instances but not in others, it is possible that you will not receive full credit from the Autograder. We have provided you with sample correct output based on the public data set, but we will also be testing your implementations against a second, hidden data set to which you will not be given access. Points for each query will be split between performance on the public and private data sets.

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 appropriate Java application function. The SQL*PLUS CLI will provide more helpful error messages than the JRE (Java Runtime Environment), making it easier to debug your solutions. In addition, it may be easier to see individual aspects of your output or to query intermediate views through the CLI.

In order to meet the runtime requirements (see below), you will need to make sure that you do the majority of your queries' "heavy lifting" through SQL and *not* through the Java application code. For example, you should not be attempting to sort data in Java; rather, use an ORDER BY clause to offload that work to the DBMS. Understanding the tradeoffs between accesses to the DBMS and runtime is an important part of this project, so we will not be answering questions about "which approach is better"; if you are unsure, try them both out and compare the runtimes!

Some of the queries are more difficult than others; while this is sometimes reflected in the point values of the queries, it is not always. If you find yourself spending an inordinate amount of time on one query, take a step back to rethink your approach or work on a different query. 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 permitted to use any and all SQL techniques to complete the nine queries for this project, including (but not limited to): JOINs, set operations (UNION, MINUS, INTERSECT), nested queries, and views. If you create any views as part of your implementation, be sure to drop them before closing the statement at the end of the Java function; you will be deducted 4 points for each view that you

fail to drop. When testing your application, the creation and dropping of views can get complicated if you have syntax errors in queries in the same function (but subsequent to) a successful CREATE VIEW statement. Due to the exception handling mechanism, your DROP VIEW statement will not be executed, and the view will persist to 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 by executing the appropriate SQL from SQL*PLUS CLI.* You are **not** permitted to create additional tables in your implementations.

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.

You may add any helper functions to the definition of the StudentFakebookOracle class that you find useful. It is possible to complete this project successfully without doing so, however.

Runtime Efficiency

As mentioned before, we will be evaluating your implementations' efficiency by measuring the time it takes to perform the query and place the results in the appropriate data structure; we will not be measuring the time it takes to print the data. The mechanism that the Autograder uses to evaluate your runtime is identical to the one used by the time make targets to report runtimes. That being said, the runtime you see from these targets when running on CAEN is not guaranteed to be identical to the time reported by the Autograder.

All of the make targets have a built-in 120-second (2-minute) timeout; any queries that take longer than this to execute will automatically terminate. Such queries will receive a 0 on the Autograder. Generally, queries that take too long will produce no output; queries with errors, on the other hand, will produce non-empty output containing default no-content output.

Based on your runtime measurements, points will be *deducted* from your output score. Each query, as listed below, has a runtime threshold. Queries that run in less time than the specified threshold will receive full credit (i.e. no deductions). Queries that run in more time than the specified threshold will receive 1 point of deduction for every threshold amount (or portion thereof) it runs slow. For example: if a query has a threshold of 2 seconds, this would be the point breakdown:

[0 − 2) seconds = 0 points deducted

[2,4) seconds = 1 point deducted [4,6)

seconds = 2 points deducted

...

You cannot earn less than 0 points on a query; queries that terminate in under 2 minutes but exceed the specified threshold by too much will earn 0 points, even if the output is correct. Queries

that do not terminate in under 2 minutes will not have their runtime measured and will receive 0 points.

For reference, here are the runtimes (as reported by the Autograder) for the official instructors' solution:

Query	Public Runtime	Private Runtime
Query 1	0.142 seconds	0.147 seconds
Query 2	0.087 seconds	0.074 seconds
Query 3	0.220 seconds	0.206 seconds
Query 4	0.148 seconds	0.155 seconds
Query 5	0.221 seconds	0.130 seconds
Query 6	6.548 seconds	3.048 seconds
Query 7	0.159 seconds	0.140 seconds
Query 8	0.160 seconds	0.163 seconds
Query 9	0.151 seconds	0.148 seconds

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 month, 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 (10 points)

Public: 5 points • Private: 5 points • Runtime Threshold: 1 second

———————————————————————————

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; again, if there are ties, report all tied names in alphabetical order. You can assume that there is at least one user listed in Fakebook's data set, and the first name is a required field of the Users table.

Query 2: Lonely Users (10 points)

Public: 5 points • Private: 5 points • Runtime Threshold: 1 second

———————————————————————————

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 (10 points)

Public: 5 points • Private: 5 points • Runtime Threshold: 1 second

———————————————————————————————————————

Query 3 asks you to identity 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 (10 points)

Public: 5 points • Private: 5 points • Runtime Threshold: 1 second

———————————————————————————————————————

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 photo's Fakebook link, the ID of the album containing the photo, 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: Match Maker (15 points)

Public: 8 points • Private: 7 points • Runtime Threshold: 1 second

———————————————————————————————————————

Query 5 asks you 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 (15 points)

Public: 8 points • Private: 7 points • Runtime Threshold: 15 seconds

———————————————————————————————————————

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. 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 should be returned; list the user with the smaller ID first 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 smaller ID first; if pairs are still tied, list the pair with the smaller larger ID first.

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.

Query 7: Event-Heavy States (10 points)

Public: 5 points • Private: 5 points • Runtime Threshold: 1 second

———————————————————————————————————————

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 that/those states. You can assume that there is at least 1 Fakebook event.

Query 8: Oldest and Youngest Friends (10 points)

Public: 5 points • Private: 5 points • Runtime Threshold: 1 second

———————————————————————————————————————

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. If two friends of

that user 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 (10 points)

Public: 5 points • Private: 5 points • Runtime Threshold: 1 second

———————————————————————————————————————

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 bee 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).

Compiling, Running, and Testing

To do anything with your Java application, you must build the following folder/file structure on CAEN machines (because of the Oracle connection restrictions). Create your project root directory using any folder name; this is where the **Makefile** goes. Inside the project root directory, create a subdirectory and name it project2 (all lower-case); this is where all of the .java files and the JAR file goes. The sample solution text file can go anywhere. **If your code is not structured like this, the Makefile will not work.**

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 and redirect the output to a file as desired. To run a single query to view the output, run make query[N] where [N] is the query number; redirect output (or don't) as you wish.

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 time[N] 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`.

Submitting

The only deliverable for Project #2 is `StudentFakebookOracle.java`. After creating a team, you may submit this file to the AutoGrader, **without compressing or modifying file name**. Each team will be allowed 3 submissions per day with feedback; any submission made in excess of those 3 will be graded, but the results of those submissions will be hidden from the group.

This project in particular takes a lengthy amount of time to grade (the instructors' solution took about 8 minutes; other implementations may take significantly longer). **Please do not make submissions to the AutoGrader right after your last submission, before the feedbacks/results are provided to you.** Please be patient and only contact the staff with concerns if you have waiting more than 25 minutes without receiving an email since your submission was taken off the queue.

Your **highest-scoring** overall submission will be accepted, but you will be penalized according to the course late policy if you submit at all during the late period, even if we accept an earlier, on-time submission as your final score. In addition, we will take your **last** submission and run it through the AutoGrader without any other load after the late deadline has passed; this is to make sure that you did not receive deductions for slow performance that were due to factors other than your implementation. **Your final score will be the larger of your best AutoGrader score and the score of this no-load post-deadline evaluation.**

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. Future 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 small 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 an instance of a variable of type `Foo`, you would write `Foo foo = 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 firstletter lower-case (i.e. long) and the reference version is first-letter upper-case (i.e. Long). This has 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 the same pointer 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. You can also use .compareTo() if the class implements the Comparable interface, but you should not need this aspect of the language for Project #2.

Java strings are references and are declared with a capital letter String. To create a new string, you can type String str = ''ABC'' without the new keyword.

Java has automatic string conversion built in to 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 an 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 therefrom.

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:

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

The reason is that the reuse of stmt to generate the results stored in rst2 causes rst to close, and then the attempt to access its data will throw an exception. If you find that you want to do something akin to this, 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.

Additional Java Libraries

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

The ROWNUM Pseudo-variable

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 a pseudo-variable that facilitates this desire, but it can be incredibly fidgety. This pseudo-variable is ROWNUM.

The was ROWNUM works is quite simple, but not always intuitive. After the FROM clause is evaluated (meaning all JOINs are completed and winnowed based on the ON clauses), each row is given a monotonically-increasing integer starting at 1; this value is stored in a psuedo-column 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 is supposed to find the users named "Bob" with the 10 smallest IDs:

```
SELECT User_ID
FROM Users
WHERE First_Name = 'Bob ' AND ROWNUM <= 10
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:

```
SELECT User_ID
FROM (
    SELECT User_ID
    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; now, when the row numbers are applied, we're in control of the order in which they are applied. We can then filter using the ROWNUM pseudo-variable and we get what we want.

The ROWNUM pseudo-variable does not error out if there aren't "enough" rows to return, as it behaves just like any other field-based conditional.

It is possible to complete this project and earn full credit without using the ROWNUM pseudo-variable.