**PS 2: Code-reading and design questions**

**The Table and Column classes**

1)

- The Table object's open() method:
    a) checks if the table is already in the in-memory cache of open tables ('t**ableCache**'). If it is, it uses the database handle and column information from the cached table.
    b) if the table is not in the cache, it retrieves the column information from the catalog (metadata).
    c) configures the database handle for the underlying Berkeley DB database and opens a handle to it.
    d) adds the table to the in-memory cache of open tables.
- If we try to call this method for a table that doesn't exist, the method return '**OperationStatus.NOTFOUND**'.
- To deal with this return value, our code should handle it appropriately based on the specific use case. E.g, we can print an error message indicating that the table doesn't exist, and then handle the error in our application logic.

2) The method that allows us to determine whether a table has a primary key is '**primaryKeyColumn()**'. This method returns the primary key column for the table if one exists or 'null' if there is no primary key defined for the table.
3) To obtain the columns associated with the table, we can use the '**getColumn(int i)**' method. This method allows us to get the column at the specified index, with the leftmost column having an index of 0. To obtain the leftmost column in the table, we would use '**getColumn(0)**'.
4) The method that allows us to determine the data type of the column is '**getType()**'. It returns an integer representing the data type of the column. In the Column class, the possible return values are defined as constants:
    - Column.INTEGER (0) for INTEGER columns.
    - Column.REAL (1) for REAL columns.
    - Column.CHAR (2) for CHAR columns.
    - Column.VARCHAR (3) for VARCHAR columns.
5) The method that allows you to determine the length of the values that will go in that column is **getLength()**. It returns an integer representing the length of the column. For VARCHAR columns, it returns the maximum number of bytes allowed for values in that column. For other column types, it returns the exact number of bytes required to represent the values.

**The SQLStatement class and its subclasses**

6) '**numTables()**' returns the number of tables specified in the SQL statement.
7) '**numColumns()**', '**numColumnVals()**', '**numWhereColumns()**'

**Marshalling**

8) For the primary key, we should use the offset '**-2**', which is represented by '**IS_PKEY**'

9) For a NULL value, we should use the offset '**-1**', which is represented by '**IS_NULL**'.
10) For the "Foo" table:
    - INSERT INTO Foo VALUES (1, 'hello'):
        - Key: The key will contain the primary key value, which is 1 in this case.
        - Value: The value will consist of offsets for two columns (a and b) and the column values.
        - Offsets: [IS_PKEY, 4] (offset for the primary key, offset for value start)
        - Column Values: [1, 'hello']
    - INSERT INTO Foo VALUES (2, NULL)
        - Key: The key will contain the primary key value, which is 2 in this case.
        - Value: The value will consist of offsets for two columns (a and b) and the column values.
        - Offsets: [IS_PKEY, IS_NULL] (offset for the primary key, offset for NULL value)
        - Column Values: [2, null] (null value for column b)

11) For the "Bar" table:
    - INSERT INTO Bar VALUES (1, '1234', 'hello', 12.5):
        - Key: The key will contain the primary key value, which is '1234' (the primary key column) in this case.
        - Value: The value will consist of offsets for three columns (a, b, c) and the column values.
        - Offsets: [IS_NULL, IS_PKEY, 6] (offset for NULL value, offset for the primary key, offset for value start)
        - Column Values: [1, '1234', 'hello', 12.5]
    - INSERT INTO Bar VALUES (2, '4567', 'wonderful', NULL):
        - Key: The key will contain the primary key value, which is '4567' in this case.
        - Value: The value will consist of offsets for three columns (a, b, c) and the column values.
        - Offsets: [IS_NULL, IS_PKEY, IS_NULL] (offset for NULL value, offset for the primary key, offset for NULL value)
        - Column Values: [2, '4567', 'wonderful', null] (null value for column d`)
12) When working with the DataOutputStream class, we will use the following methods for writing offsets and column values:
    - writeInt(int v): Used to write integer values.
    - writeUTF(String str): Used to write UTF-8 encoded strings.
    - writeFloat(float v): Used to write floating-point values.
    - writeDouble(double v): Used to write double-precision values.

**Unmarshalling**

13) To retrieve a particular column value from a row:
    - We have a key/value pair (presumably in the form of DatabaseEntry objects, key and value).

- First, we would typically retrieve the value from the value part of the key/value pair.
- Next, we will create a RowInput object using the byte array from the value portion. The RowInput class allows us to read values from a byte array.
- Finally, we can use the RowInput methods like readBooleanAtOffset(), readIntAtOffset(), readDoubleAtOffset(), or readBytesAtOffset() to read values for specific columns at the appropriate offsets.

14) We would use various methods from the RowInput class based on the data type of the column we are reading. For example:
    - To read a boolean value, you would use readBooleanAtOffset().
    - To read a byte, you would use readByteAtOffset().
    - To read a short (two-byte integer), you would use readShortAtOffset().
    - To read an int (four-byte integer), you would use readIntAtOffset().
    - To read a double, you would use readDoubleAtOffset().
    - To read VARCHAR or CHAR values, you would use readBytesAtOffset() and specify the number of bytes to read.
15) When unmarshalling a VARCHAR attribute, we typically have to know the length of the VARCHAR. This information is often stored as part of the marshalled data.
    We would use the offset and the format we used during marshalling to determine where the VARCHAR data begins and ends in the value portion of the key/value pair. This offset is part of the marshalling scheme we designed.
    Once we know the start and end offsets of the VARCHAR data, we can calculate the length and use it to read the appropriate number of bytes from the value portion.

**Miscellaneous**

16) The provided code in the execute() method of the InsertStatement class uses objects from the InsertRow class to:
    - Open the specified table.
    - Perform error checks on the InsertStatement.
    - Adjust the values to be inserted.
    - Create an InsertRow object representing the row to be inserted.
    - Call the marshall() method on the InsertRow object to format the data for insertion.
17) In the rest of the execute() method in the InsertStatement class, you will need to perform the following tasks:
    - Use Berkeley DB methods to open the database cursor.
    - Use the cursor to insert the marshalled row (key/value pair) into the table's database.
    - Check for errors during the insertion process and handle exceptions appropriately.
    - Print a success message if the insertion is successful.

    The following Berkeley DB methods will be required:
    - Opening a cursor to access the database.
    - Using the cursor's put method to insert the key and value into the database.

- Handling exceptions related to Berkeley DB operations, such as DatabaseException or DeadlockException.

18) In executing a SELECT * statement, we will make use of the printAll() method for table iterators. This method invokes the getColumnVal(int colIndex) method from the TableIterator class to retrieve column values. We will need to implement the getColumnVal(int colIndex) method to retrieve the column values for the SELECT * statement.