

CSI 410. Database Systems – Spring 2021

Programming Assignment I

The total grade for this assignment is 100 points. The deadline for this assignment is **11:59 PM, March 3, 2021**. *Submissions after this deadline will not be accepted.* Students are required to enter the UAlbany Blackboard system and then upload a .zip file (in the form of [first name]_[last name].zip) that contains the Eclipse project directory and a document succinctly describing:

- any missing or incomplete elements of the code
- any changes made to the original API
- the amount of time spent for this assignment
- suggestions or comments if any

No submission of the above document will lead to a loss of 5 grade points. Also, please add *comments* in your code. Code with (almost) no comments may result in a loss of 1 grade point per method to implement.

In this programming assignment, you need to implement several classes for representing both relational and non-relational data. You first need to install and run Eclipse on your machine and import the “hdb_data” project (see Appendix A). Please generate an API document (see Appendix B) and then take a look at that document as well as the source code to familiarize yourself with this assignment. This assignment provides you with a set of incomplete classes. To see these classes, choose the “src” source folder and then either the “hdb.data.relational” package or the “hdb.data.nonrelational” package. You will need to write code for these classes. Your code will be graded by running a set of unit tests (see the unit tests in the “test” source folder which use JUnit¹) and then examining your code. For details of running these tests, refer to Appendix A. Note that passing unit tests does NOT necessarily guarantee that your implementation is correct and efficient. Please make sure that your code would NOT cause problems even in situations not covered by the unit tests. If you have questions, please contact the TA(s) or the instructor. The remainder of this document describes the components that you need to implement.

Part 1. Relation Schema (40 points)

A `RelationSchema` represents the schema of a relation. A `RelationSchema` contains an array `attributeNames` which stores the names of attributes (e.g., { "ID", "Name" }) and another array `attributeTypes` which contains the types of the attributes (e.g., { `Integer.class`, `String.class` }). In this part, you need to complete the following methods of the `RelationSchema` class:

- `size()`: returns the number of attributes in the `RelationSchema`.
- `attributeName(int attributeIndex)`: returns the name of the attribute specified by `attributeIndex`. For example, if array `attributeNames` is set to { "ID", "Name" }, then `attributeName(0)` and `attributeName(1)` must return "ID" and "Name", respectively.

¹<http://junit.org>

- `attributeType(int attributeIndex)`: returns the type of the attribute specified by `attributeIndex`. For example, if array `attributeTypes` is set to `{ Integer.class, String.class }`, then `attributeType(0)` and `attributeType(1)` must return `Integer.class` and `String.class`, respectively.
- `attributeIndex(String attributeName)`: returns the index of the attribute specified by `attributeName`. For example, if array `attributeNames` is set to `{ "ID", "Name" }`, then `attributeIndex("ID")`, `attributeIndex("Name")`, and `attributeIndex("Address")` must return 0, 1, and null, respectively.
- `save(String fileName)`: writes the `RelationSchema` (on which this method is called) to the file specified by `fileName`. This method needs to create a `FileOutputStream` using `fileName` and then an `ObjectOutputStream` that writes to that `FileOutputStream`. The `RelationSchema` on which this method is called needs to be written to the `ObjectOutputStream` using the `writeObject(Object)` method. If your code throws a `java.io.NotSerializableException`, address that problem after understanding the basics of Java object serialization. Details of serialization can be found at: <https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/index.html>.

Please make sure that your code passes all of the 5 unit tests in `RelationSchemaTest`. These unit tests verify the 5 methods mentioned above using one instance of `RelationSchema` containing 2 attributes ("ID" and "Temperature" of the `Integer` and `Double` types, respectively) and another instance containing 3 attributes ("ID", "Name", and "Address" of the `Integer`, `String`, and `String` types, respectively).

Part 2. Tuple (40 points)

A `Tuple` is a record in a relation. Each `Tuple` contains the values of certain attributes representing an entity. For example, a tuple `[123, "John"]` in a relation with attributes "ID" and "Name" would represent a person whose ID is 123 and whose name is "John". Each `Tuple` has an array `attributeValues` which stores the values of attributes defined in the corresponding `RelationSchema`.

In this part, you need to implement the following methods in `Tuple.java`:

- `setAttribute(int attributeIndex, Object o)`: sets the value of the attribute specified by `attributeIndex` to `Object o`. For example, `setAttribute(0, 123)` and `setAttribute(1, "John")` would set `attributeValues[0]` to 123, and `attributeValues[1]` to "John", respectively. If the specified attribute value (`o`) is not compatible with the type of the attribute, then the method needs to throw a `TypeException` exception. In other words, when the `RelationSchema` has an array of types `{ Integer.class, String.class }`, `setAttribute(0, "John")` must throw a `TypeException` exception since the attribute at index 0 is of the `Integer` type and thus cannot be set to "John". Consider using the `Class#isInstance(Object)` method to check type compatibility.
- `writeAttributes(ObjectOutputStream out)`: writes the attribute values of the `Tuple` to the specified `ObjectOutputStream`. This method needs to call the `write(Object o, ObjectOutputStream out)` method to write each attribute value to the `ObjectOutputStream`. As explained below, Java object serialization tends to incur high space overhead. The purpose of `writeAttributes(ObjectOutputStream out)` is to reduce this overhead through custom serialization.
- `write(Object o, ObjectOutputStream out)`: writes the specified `Object` to the specified `ObjectOutputStream`. Writing `Object o` to the `ObjectOutputStream` using the

`ObjectOutputStream#writeObject(Object)` method (which uses the standard Java object serialization) may incur high space overhead. Therefore, `write(Object o, ObjectOutputStream out)` needs to call (i) `ObjectOutputStream#writeInt(int)` instead if the given `Object` is of the `Integer` type, (ii) `ObjectOutputStream#writeDouble(double)` instead if the given `Object` is of the `Double` type, and (iii) `ObjectOutputStream#writeObject(Object)` in the other cases.

- `read(Class<?> type, ObjectInputStream in)`: reads an object from the specified `ObjectInputStream`. This method needs to call (i) `ObjectInputStream#readInt()` if the specified type is `Integer.class`, (ii) `ObjectInputStream#readDouble()` if the specified type is `Double.class`, and (iii) `ObjectInputStream#readObject()` otherwise.

Please verify your code using the tests in `TupleTest`. Your code can pass the `writeAttributes` test in `TupleTest` as long as `writeAttributes(ObjectOutputStream out)` is correctly implemented (even if `write(Object o, ObjectOutputStream out)` and `read(Class<?> type, ObjectInputStream in)` are not fully implemented). The following result from the `writeAttributes` test (when `write(Object o, ObjectOutputStream out)` and `read(Class<?> type, ObjectInputStream in)` are not completed) shows the difference between standard Java object serialization and our custom serialization (when it is partially implemented) given a `Tuple` containing just one `Integer` and one `Double` value.

```
size of tuple [1, 5.0] (standard Java serialization): 538
size of tuple [1, 5.0] (custom serialization): 134
```

The `writeRead` test in `TupleTest` requires complete implementation of `write(Object o, ObjectOutputStream out)` and `read(Class<?> type, ObjectInputStream in)`. When Part 2 is completed, the `writeRead` will produce the following result (there is still 6 bytes of additional data written in addition to the 4-byte `int` and 8-byte `double` values):

```
size of tuple [1, 5.0] (standard Java serialization): 538
size of tuple [1, 5.0] (custom serialization): 18
```

Part 3. Non-Relational Data (15 points)

In this part, you need to complete the `CollectionSchema` and `DataObject` classes in the `hdb.data.nonrelational` package. These two classes are similar to `RelationSchema` and `Tuple`, but their main difference is that they support non-relational data. In detail, while all `Tuples` in the same relation have the same set of attributes, `DataObjects` in the same collection can have different attributes. Furthermore, a `DataObject` may have a hierarchical structure. For example, for a `DataObject`, the value of the "Name" attribute can be another `DataObject` having "FirstName" and "LastName" as its attributes. In this case, given the former `DataObject`, "Name.FirstName" and "Name.LastName" would represent the first name and second name of a person, respectively.

The `CollectionSchemaTestBasic` and `DataObjectTestBasic` classes in the `test` source folder are designed to verify the correctness of the code when the `CollectionSchema` and `DataObject` classes do not fully support hierarchical data representation. The `CollectionSchemaTestAdvanced` and `DataObjectTestAdvanced` tests check whether or not `CollectionSchema` and `DataObject` can represent data in a hierarchical manner.

Appendix A. Installing Eclipse and Importing a Java Project

1. Visit:

<http://www.eclipse.org/downloads/>

2. From the web site, download the eclipse installer (those for Linux, Windows, and Mac OS X are available) and then choose "Eclipse IDE" and install it.

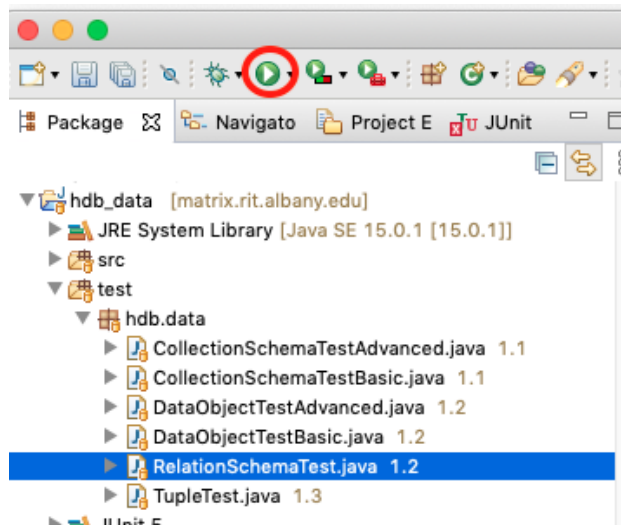


Figure 1: Eclipse

3. After finishing installation, start Eclipse.
4. When Eclipse runs for the first time, it asks the user to choose the workspace location. You may use the default location.
5. In the menu bar, choose “File” and then “Import”. Next, select “General” and “Existing Projects into Workspace”. Then, click the “Browse” button and select the “hdb_data.zip” file contained in this assignment package.
6. Once the project is imported, you can choose the “hdb.data” package in the “test” source folder. Then, you can run one among `RelationSchemaTest.java`, `TupleTest.java`, `CollectionSchemaTestBasic.java`, `CollectionSchemaTestAdvanced.java`, `DataObjectTestBasic.java`, and `DataObjectTestAdvanced.java` by clicking the icon highlighted in Figure 1.

Appendix B. Creating API documents using javadoc

One nice feature of Java is its support for “documentation comments”, or “javadoc” comments, which you can use to automatically produce documentation for your code. Javadoc comments start with “/”**. Inside a javadoc comment, there are some special symbols, like `@param` and `@return`.

You can create HTML-based API documents from the source as follows:

1. Click the “hdb_data” project icon in the Navigator or Project Explorer window.
2. Select “Generate Javadoc” from the “Project” menu.
3. In the “Generate Javadoc” dialog box, press the “Finish” button.

As it runs, it tells you that it’s generating various things. When it is finished, a few new folders should appear in your project: `doc`, `doc.resources`, and so on. See what got generated (to open the newly created HTML documentation files in a web browser window, just double-click them; you can start with “`index.html`”).