COMP 6411
Final Project

In the final project, your job will be to work with both and imperative and functional language. Given the limited time frame, the project has been designed so that the two languages can be used and implemented more of less separately. So the idea is to work on the functionality of the imperative component and then move to the additional functionality provided by the functional module.

Basically, you are going to design and implement an embedded database engine. In contrast to a standalone server, an embedded database system is one that is linked or attached directly to another application (that another programmer would develop). You can think of it as a library of database methods that provides a powerful storage and retrieval API. As you read the description below, the idea will become clearer.

Please don't be frightened by the length of the project description. It is NOT ten times bigger than the assignments. I have simply provided as many details and examples as possible so that you will have a good sense of how everything should be done.

The Slice DBMS (70 points)

You will be creating a DBMS library that we will call **Slice**. It will basically consist of an API (Application Programming Interface) that provides access to the DB methods, plus an indexing mechanism. Wow, you say, that's easy. Well, as people often say, "the devil is in the details."

Let's look at how someone (me, for example) might use your system to create a Slice database. The Java code on the next page will serve as our example (sorry, YOU can't write it in Java). You will notice a few things. First, the code for the library is located in a package called SliceDBMS. You will also want to separate your functionality into modules, packages, etc. In other words, we don't want to see one large source file.

My code is in the **main** method. I begin by defining a schema for the database. I must do this since eventually the Slice DBMS must write records to disk and it must know what it is trying to write (size, format, etc.). In our case, we will assume that Slice can handle strings, ints, and doubles. Slice should define constants for these types. The schema itself consists of a series of Slice Fields that associate a type with a column name.

```
// SCHEMA DEFINITION example
package sliceDBMS;

public class Test_Create_DB {

	public static void main(String[] args) {

		SliceField[] schemaElements = {
			new SliceField("cust", SliceDB.INT),
			new SliceField("name", SliceDB.STRING),
			new SliceField("age", SliceDB.INT),
			new SliceField("phone", SliceDB.STRING),
			new SliceField("address", SliceDB.STRING)
		};

		SliceEnv env = new SliceEnv();
		env.createDB(
			"CustDB",
			schemaElements,
			"cust"
		);
	}
}
```

Now, once the schema is defined, I will use it to define a new database. However, I need a way to create the database. For this, we use the SliceEnv object. This is our DBMS environment. Basically is it our interface into the Slice system and is used primarily to create, open, and close databases. Once we have a new environment, we then create a database by providing three arguments: a database name, a schema, and an (optional) index column. Slice databases may have an index in order to support fast lookups.

Finally, you should note that, in the Slice embedded DBMS, a Slice database is essentially equivalent to a single table. If I want to create many tables, each would be housed in its own Slice DB. This is important to remember.

Storing data in Slice

So what do you put in Slice? As you may recall, it's possible in Java (and other languages) to simply serialize data or objects to disk. Basically, the Java run-time just writes objects out to disk as a sequence of bytes. This is great for simple storage requirements. But it is not good for us since we must ensure that two languages will be able to read/write the files.

Instead, we will treat our records as simple objects. In short, the programmer will create Slice Records that the Slice DBMS can then manipulate. When it is time to write data to disk, we can simply do that by outputting a "|" delimited text file, one record per row.

So what is a Slice Record? Your DBMS must have the ability to store and return fields and to convert these fields into values that can be stored to disk. Again, let's look a simple piece of code that builds upon our previous example.

```
//RECORD INSERTION example

package sliceDBMS;

public class Test_Create_Records {

      public static void main(String[] args) {

            SliceEnv env = new SliceEnv();
            SliceDB custDB = env.open("CustDB");

            SliceRecord custRecord = custDB.createRecord();

            custRecord.setString("name", "Joe Smith");
            custRecord.setInt("age", 43);
            custRecord.setString("address", "Montreal");

            custDB.set(custRecord);

            env.close("CustDB");
      }
}
```

Above, you will see a program that creates a new record and adds it to the Customer database. I begin by creating a new environment and retrieving the existing Customer database (it's empty right now). Next, I use the database object to create an empty record for me.
Why don't I just do something like **`new SliceRecord( ).`** If I did this, the new record would have no knowledge of the schema created for the Customer database. This would make it almost impossible to do anything useful with records. By using the database to create records, you should be able to do something about this.

Now, once we have a Slice Record, we will use the various "set" methods to add values to this new record. When we are ready, we add the record to the database. Note that this method is called "set", not "add". Why? If a given customer already exists, the set method will replace the old record with the new record. If it doesn't exist, the set method will add it. Basically, if I want to update a record, I must provide an existing key value. If I do not provide a key, then the record will be added as a new customer (cust name does not have to be unique).

In this case, I did not specify the cust number, so a new record would be added. Any other unspecified columns (e.g., "phone") should simply be left blank.

Appending data in Bulk mode

In addition to adding a record, it will be important to bulk load or append a large number of records. This will be useful for testing purpose. For this you will provide a **load** method for the SliceDB object. It will simply take one argument – the name of a text file that contains the records (the format of the records in the test file must match the schema you have defined, of course). When the load method is called, the records added to the database file. Note: any existing records are deleted before the load is done. So after the load, only the newly inserted records will remain.

To generate the records, you will use table generator. Application. This is discussed later in the assignment.

Simple Lookups

Okay, so now we have data in the system. How do we get it out? For this we do a "get". In Slice, we use the indexed column for retrievals. In our running example, this is the "cust" field. So all I have to do is indicate the customer number and Slice will return a Slice Record with the appropriate data.

```
// RECORD LOOKUP example

package sliceDBMS;

public class Test_Get_Record {

	public static void main(String[] args) {

		SliceEnv env = new SliceEnv();
		SliceDB custDB = env.open("CustDB");

		SliceRecord custRecord = custDB.get(21);

		System.out.println(
			"Name: " + custRecord.getString("name"));
		System.out.println(
			"Age: " + custRecord.getInt("age"));
		System.out.println(
			"Address: " + custRecord.getString("address"));

		env.close("CustDB");
	}
}
```

The code above illustrates this case. It is actually quite simple. Once the database is open, we call the `get` method and specify the number of the customer. Once the record is returned, we

may invoke the various "get" methods to return the appropriate values. In this case, we just print out the three values. And that's it.

Working with more than one table

The functionality provided so far would be enough to produce a fairly useful database library. However, we have assumed that users never need to work with more than one table. It would be nice to address this shortcoming.

We will do this by introducing a simple *join* operation. In our case, it will be a "Natural" join. Basically, this means that our join will always be performed on a "matching" column found in both tables. For this to work, the columns in the two tables must have the same name and be of the same type (trust me, this simplifies things).

The code sample on the next page shows how I might use Slice's join capability. Again, note that an ArrayList is being used since there can be multiple records in the result. You will also see that the join is specified by invoking the join method from one database object, and using the second database object as the join target. Once the result is returned, we can again iterate through the ArrayList and process the results.

```
// JOIN example

package sliceDBMS;

import java.util.ArrayList;

public class Test_Join {

	public static void main(String[] args) {

		SliceEnv env = new SliceEnv();
		SliceDB custDB = env.open("CustomerDB");
		SliceDB salesDB = env.open("SalesDB");

		ArrayList<SliceRecord> joinResult =
			custDB.join(salesDB);

		for(int i =0; i < joinResult.size(); i++){
			String name =
				joinResult.get(i).getString("Customer");

			Double amount =
				joinResult.get(i).getDouble("Amount");

			System.out.println(
				"Sales for " + name + ": " + amount);
		}

		env.close("CustomerDB");
	}
}
```

What might not be so obvious is that Slice must be able to dynamically create a new schema for the joined result. After all, the two databases may only have one field in common (i.e., the index column). So, in order to be able to process the results, the new schema must include the fields of both databases.

Basic Queries

We also want to be able send VERY simple queries to the database. A "real" DBMS uses a query language like SQL. Clearly, you can't create a query language for your project. Instead, you will provide a simple programmatic interface for running a query. A Query will effectively have three parts:

1. A list of attributes to display in the results. If you are familiar with SQL, these are the columns listed in the SELECT clause.
2. A target database. This is effectively the FROM clause in SQL. Note that our queries are only performed on one table. You do not have to do Joins in these queries.

3. A condition that restricts the query. In SQL, this would be defined in the WHERE clause. In practice, conditions can be very complex. You are going to support something much, much simpler. Specifically, your conditions will be of the form: <column_name, operator, literal>. These are defined as follows:
    a. Column_name: simply the name of a column from the target table
    b. Operator: One of three choices: <, >, =
    c. Literal: a value that will be compared to the column value

Let's say, for example, that we want to display all customers older than 25. In this case, our condition would be: (age, >, 25). A test for all customers name "Smith" would be (name, =, "Smith") (Note that comparisons are case sensitive so we just need an exact match on strings.) While these queries are defined in a simple way, they can be used to do more interesting things. For example, in order to update a customer, you must know the customer number. It is not likely that would actually know this. Instead, you could run a query that returns (name = "Joe Smith", then get the cust number and update the record.

For the project, the idea is that you would then run these queries and display the results. The example on the next page illustrates the process. Note how the query components are constructed in a simple compositional style.

```
// QUERY example

package sliceDBMS;

import java.util.ArrayList;

public class Test_Query {

      public static void main(String[] args) {

            SliceEnv env = new SliceEnv();
            SliceDB custDB = env.open("CustDB");

            // create list of columns to display
            ArrayList<String> columns = new Arraylist<String>();
            columns.add("name");
            columns.add("age");

            // create condition
            ArrayList<SliceCondition> condition =
                  new Arraylist<SliceCondition>();
            condition.addColumn("name");
            condition.addOp( SliceOP.EQ );
            condition.addLiteral("Joe Smith");


            SliceQuery custQuery =
                  new SliceQuery(columns, "CustDB", condition);

            ArrayList<SliceRecord> queryResult =
                  custDB.query(custQuery);

            // display query result
            System.out.println("Customer  Age");
            for(int i =0; i < queryResult.size(); i++){
                  String name =
                        queryResult.get(i).getString("name");

                  Double age =
                        queryResult.get(i).getInt("age");

                  System.out.println(name + "  " + age);
            }

            env.close("CustDB");
      }
}
```

Disk Files

Unlike a real database, you are not going to store records in some sort of proprietary binary format. Instead they will be stored as delimited text files, with the "|" symbol as the delimiter. Database files will be store as tableName.slc to indicate a slice database file. Any bulk append files will be stored as tableName.apd. Again, these are just text files that can be read with any text editor.

Reports

The main part of your DBMS will be constructed using the imperative language of your choice. The second portion of the project will use the functional language. The report module(s) is distinct from the main DBMS system. In short, reports will be run on the text files that constitute the database and the results displayed to the screen.

Note that the disk files must be updated in order for this to work. When the menu option for the report(s) is selected, the main program should first update the disk file with the latest contents of the table. You don't have to do anything fancy here. Just delete the current disk file and write out a new one. Once the disk file is read, then the menu program can call the new application with a "shell" method. Every language has a mechanism to start another executable. Typically, it is something like:

```
os.system("C:\\mypath\\report1.exe");
```

This will run the program and print the report to the screen.

Report 1 (10 points)

The first report will be what we call a **Group By** report. Here, the idea is to organize the output into categories and then print a summation for some column for each category. To keep things simple, we will specifically use a SalesDB table for this report (schema defined later) and we will be summarizing sales on cust number. So the idea is that we would produce a report that looks like the following:

Cust	Sales
1	234.05
2	43.5
3	398.45
…	
98	76.50

As you can see, each customer is associated with a total sales, which is the summation of the sales for each individual customer. Each customer will only appear once in the list.

Report 2 (20 points)

The second report is a little different. It will be used to produce a display of purchases for senior citizens. In our case, seniors will be just those customers who are at least 60 years old. For this you will need to combine the records from the CustDB, SalesDB, and OrderDB. The idea is to simply produce a report that shows the items purchased by each customer on a given order. The report would look like the following:

Customer	Order#	Date	Items
Samson Bowman	17	20/3/2014	shoes, socks, milk
Samson Bowman	34	19/5/2014	gum, sandals, butter, pens, pencils
Paloma Deleon	41	6/1/2014	computer
…			
Cynthia Woodward	121	26/5/2014	gum, chips

Note that a customer name can appear multiple times since a customer can make many orders.

Schemas

For testing purposes, you will need to (1) create new schemas dynamically and (2) utilize existing schemas. Pre-defined schemas will be need to (1) demonstrate bulk loading capabilities and to create realistic Haskell reports. Specifically you will need to support the CustDB, SalesDB, and OrderDB tables. (Again, you make have to create additional schemas dynamically during the demo, and populate them with data).

To quickly generate these records, you will use a simply, free application called **spawner**. This can be downloaded from:

http://sourceforge.net/projects/spawner/

Using the application is very easy. You can simply define your columns names, then indicate the properties for that column (as per the schemas below). If you update a column's properties, make sure to press the "save field" button again. Once you are happy with the schema, you can save the schema definition to a text file so that you can use it later. On the output tab, you can indicate several other things, including the name of the output file, the type of output file, and the number of records. You are going to create "delimited" files, using the "|" character. Again, whenever you make any changes, remember to save the updates to the appropriate schema file. Once this is done, you can generate the new output file by pressing the "spawn" button. That's it.

The format of schemas that you need to generate are listed below:

CustDB schema (100 customers)
cust: integer sequence from 1 to 100
name: Full name (e.g., Cynthia Woodward)
age: int in the range 20 to 80
phone: random phone number
address: random City value

Example
1|Samson Bowman|68|789-174-4831|Woodruff
2|Zelda Graves|31|454-511-7072|Rock Island
3|Noah Hensley|43|460-988-6983|Chattanooga
4|Noelle Haynes|40|060-857-2412|Kona
5|Paloma Deleon|48|147-915-2319|Mission Viejo

SalesDB schema (1000 orders)
order: integer sequence from 1 to 1000
cust: integer in range 1 to 100
date: random date in d/m/y format from range 1/1/2014 to 8/07/2014
total: random float value in range 0.50 to 150.00, using two decimal places

Example
1|3|20/3/2014|69.46
2|4|25/4/2014|38.32
3|5|17/7/2014|32.75
4|9|5/1/2014|97.13
5|5|9/6/2014|78.92

OrderDB schema (10000 order items)
order: integer in range 1 to 1000
item: random choice from following set: bread | butter | candle | chips | computer | flowers | gum | milk | pen | pencil | phone | pop |sandals | shoes | socks | tablet

Example
2|gum
4|sandals
3|pen
1|gum
2|pen
3|chips
1|pop
5|chips

Main Menu

For testing and demo purposes, your main application (written in the imperative language) must provide a simple menu – much like assignment 1. It simply has to provide the functions listed in the assignment (you can add others if you like):

Create database
Update Record
Add record
Bulk load
Display Join
Run Query
Report 1
Report 2

Once a menu option is run and shows its results, the menu will be displayed again.

Submisson

You will of course submit your source code via Moodle. The code will be submitted as **project_lastName_firstName_ID.zip**. You should also include a README text file with any instructions or comments about problems or bugs. This will make things easier for everyone. Demos will also be done (time slots and final deadlines to be posted soon).

That's about it. So have fun and get *Slicin'*.

Good Luck!