Lab 2 - SimpleDB

DBSys 2023-2024

Assigned: Oct 27th, 2023	Due: Dec 7th, 2023

This lab is based on MIT's 6.830 - Database Systems course. Labs initially created by Prof. Sam Madden.

1 Overview

Through a series of two labs you will create a basic database management system called SimpleDB. For this lab, you will focus on implementing the core modules required to access stored data on disk; in future labs, you will add support for various query processing operators.

SimpleDB is written in Java. We provide a set of mostly unimplemented classes and interfaces. You will write the code for these classes. We will grade your code by running a set of system tests written using JUnit. We also provide a number of unit tests, which we will not use for grading but that you may find useful in verifying that your code works.

The remainder of this document describes the basic architecture of SimpleDB, gives some suggestions about how to start coding, and discusses how to hand in your lab.

As this lab requires you to write a fair amount of code, we **strongly recommend** that you start as early as possible on it.

2 Environment Setup

Start by downloading the starter code for this lab

Because the code is written in Java, it should work under Windows as well, although the directions in this document may not apply.

We have included in Section 2.5 instruction how to load the project in Eclipse.

2.1 Getting started

SimpleDB uses the Ant build tool to compile the code and run tests. Ant is similar to make, but the build file is written in XML and is somewhat better suited to Java code. Most modern Linux distributions include Ant.

To help you during development, we have provided a set of unit tests in addition to the end-to-end tests that we use for grading. These are by no means comprehensive, and you should not rely on them exclusively to verify the correctness of your project.

To run the unit tests use the `test` build target:

```
$ cd [project-directory]
$ # run all unit tests
$ ant test
$ # run a specific unit test
$ ant runtest -Dtest=TupleTest
```

You should see output similar to:

```
 build output...

test:
    [junit] Running simpledb.CatalogTest
    [junit] Testsuite: simpledb.CatalogTest
    [junit] Tests run: 2, Failures: 0, Errors: 2, Time elapsed: 0.037 sec
    [junit] Tests run: 2, Failures: 0, Errors: 2, Time elapsed: 0.037 sec

 ... stack traces and error reports ...
```

The output above indicates that two errors occurred during compilation; this is because the code we have given you doesn't yet work. As you complete parts of the lab, you will work towards passing additional unit tests.

At different steps in the project, you can decide to run only the tests covering files that you have implemented up to that point. If you wish to write new unit tests as you code, they should be added to the test/simpledb directory.

For more details about how to use Ant, see the manual. The Running Ant section provides details about using the `ant` command. However, the quick reference table below should be sufficient for working on the labs.

Command	Description
`ant`	Build the default target (for simpledb, this is dist)
`ant -projecthelp`	List all the targets in build.xml with descriptions.
`ant dist`	Compile the code in src and package it in dist/simpledb.jar.
`ant test`	Compile and run all the unit tests.
`ant runtest -Dtest=testname`	Run the unit test named testname.
`ant systemtest`	Compile and run all the system tests.
`ant runsystest -Dtest=testname`	Compile and run the system test named testname.

If you are using a windows OS and don't want to run ant tests from command line, you can also run ant within eclipse. Right click build.xml, in the targets tab, you can see "runtest" "runsystest" etc. For example, select runtest would be equivalent to "ant runtest" from command line. Arguments such as "-Dtest=testname" can be specified in the "Main" Tab, "Arguments" text box. Note that you can also create a shortcut to runtest by copying from build.xml, modifying targets and arguments and renaming it to, say, runtest_build.xml.

2.2 Running end-to-end tests

We have also provided a set of end-to-end tests that will eventually be used for grading. These tests are structured as JUnit tests that live in the test/simpledb/systemtest directory. To run all the system tests, use the `systemtest` build target:

```
$ ant systemtest

 ... build output ...

    [junit] Testcase: testSmall took 0.017 sec
    [junit]     Caused an ERROR
    [junit] expected to find the following tuples:
    [junit]     19128
    [junit]
    [junit] java.lang.AssertionError: expected to find the following tuples:
    [junit]     19128
    [junit]
    [junit]     at simpledb.systemtest.SystemTestUtil.matchTuples(SystemTestUtil.java:122)
    [junit]     at simpledb.systemtest.SystemTestUtil.matchTuples(SystemTestUtil.java:83)
```

```
    [junit]     at simpledb.systemtest.SystemTestUtil.matchTuples(SystemTestUtil.java:75)
    [junit]     at simpledb.systemtest.ScanTest.validateScan(ScanTest.java:30)
    [junit]     at simpledb.systemtest.ScanTest.testSmall(ScanTest.java:40)

 ... more error messages ...
```

This indicates that this test failed, showing the stack trace where the error was detected. To debug, start by reading the source code where the error occurred. When the tests pass, you will see something like the following:

```
$ ant systemtest

 ... build output ...

    [junit] Testsuite: simpledb.systemtest.ScanTest
    [junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 7.278 sec
    [junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 7.278 sec
    [junit]
    [junit] Testcase: testSmall took 0.937 sec
    [junit] Testcase: testLarge took 5.276 sec
    [junit] Testcase: testRandom took 1.049 sec

BUILD SUCCESSFUL
Total time: 52 seconds
```

2.3 Creating dummy tables

It is likely you'll want to create your own tests and your own data tables to test your own implementation of SimpleDB. You can create any .txt file and convert it to a .dat file in SimpleDB's `HeapFile` format using the command:

```
$ java -jar dist/simpledb.jar convert file.txt N
```

where file.txt is the name of the file and N is the number of columns in the file. Notice that file.txt has to be in the following format:

```
int1,int2,...,intN
int1,int2,...,intN
int1,int2,...,intN
int1,int2,...,intN
```

...where each intN is a non-negative integer.

To view the content of a table, use the `print` command:

```
$ java -jar dist/simpledb.jar print file.dat N
```

where file.dat is the name of a table created with the convert command, and N is the number of columns in the file.

2.4 Working in IntelliJ IDEA

Intellij IDEA is a graphical software development environment that you might be more comfortable with working in.

2.4.1 Setting the Lab Up in Intellij IDEA

- Open Intellij and select *Import Project*.
- In the dialog, select the folder where Lab 1 files are located.
- Select *Create Project* from existing sources.
- Continue clicking next leaving everything else as default.
- Once the project is loaded, go to the Project Structure and then open the *Modules* settings. Open the *Sources* tab and right-click on the *java* folder (located inside *src*) and set it as a sources folder.
- It might be that IntelliJ also created a separate module for the src folder. Remove that module.

2.4.2 Running Indivisual Unit and System Tests

You can run tests by right-clicking on a test class. For example you might want to right-click the class TupleTest (located in "test/simpledb" folder), and then click Run 'TupleTest'.

2.4.3 Running Ant Build Targets

If you want to run ant targets in IntelliJ IDEA, you can add the Ant project by opening the *Ant* tab on the right and clicking the button with the plus symbol. In the dialog select the file *build.xml*. Now you will be able to select and run ant actions through the *Ant* menu.

2.5 Working in Eclipse

Eclipse is a graphical software development environment that you might be more comfortable with working in. The instructions we provide were generated by using Eclipse for Java Developers (not the enterprise edition).

2.5.1 Setting the Lab Up in Eclipse

- Once Eclipse is installed, start it, and note that the first screen asks you to select a location for your workspace (we will refer to this directory as $W). Select the directory containing your Lab1 folder.
- In Eclipse, select File → New → Project → Java → Java Project, and click Next.
- Enter "Lab1" as the project name.
- On the same screen that you entered the project name, select "Create project from existing source," and navigate to $W/Lab1.
- Click finish, and you should be able to see "Lab1" as a new project in the Project Explorer tab on the left-hand side of your screen. Opening this project reveals the directory structure discussed above - implementation code can be found in "src," and unit tests and system tests found in "test."

Note: that this class assumes that you are using the official Oracle release of Java. This is the default on MacOS X, and for most Windows Eclipse installs; but many Linux distributions default to alternate Java runtimes (like OpenJDK). Please download the latest Java8 updates from Oracle Website, and use that Java version. If you don't switch, you may see spurious test failures in some of the performance tests in later labs.

Running Individual Unit and System Tests

To run a unit test or system test (both are JUnit tests, and can be initialized the same way), go to the Package Explorer tab on the left side of your screen. Under the "Lab1" project, open the "test" directory. Unit tests are found in the

"simpledb" package, and system tests are found in the "simpledb.systemtests" package. To run one of these tests, select the test (they are all called *Test.java - don't select TestUtil.java or SystemTestUtil.java), right click on it, select "Run As," and select "JUnit Test." This will bring up a JUnit tab, which will tell you the status of the individual tests within the JUnit test suite, and will show you exceptions and other errors that will help you debug problems.

2.5.2 Running Ant Build Targets

If you want to run commands such as "ant test" or "ant systemtest," right click on build.xml in the Package Explorer. Select "Run As," and then "Ant Build..." (note: select the option with the ellipsis (...), otherwise you won't be presented with a set of build targets to run). Then, in the "Targets" tab of the next screen, check off the targets you want to run (probably "dist" and one of "test" or "systemtest"). This should run the build targets and show you the results in Eclipse's console window.

2.6 Implementation hints

Before beginning to write code, we **strongly encourage** you to read through this entire document to get a feel for the high-level design of SimpleDB.

You will need to fill in any piece of code that is not implemented. It will be obvious where we think you should write code. You may need to add private methods and/or helper classes. You may change APIs, but make sure our grading tests still run and make sure to mention, explain, and defend your decisions in your writeup.

In addition to the methods that you need to fill out for this lab, the class interfaces contain numerous methods that you need not implement until subsequent labs. These will either be indicated per class:

```
// Not necessary for lab1.
public class Insert implements DbIterator {
```

or per method:

```
public boolean deleteTuple(Tuple t) throws DbException {
    // some code goes here
    // not necessary for lab1
    return false;
}
```

The code that you submit should compile without having to modify these methods.

We suggest exercises along this document to guide your implementation, but you may find that a different order makes more sense for you.

Here's a rough *high-level* outline of one way you might proceed with your SimpleDB implementation:

- Implement the classes to manage tuples, namely Tuple, TupleDesc. We have already implemented Field, IntField, StringField, and Type for you. Since you only need to support integer and (fixed length) string fields and fixed length tuples, these are straightforward.
- Implement the Catalog (this should be simple).
- Implement the BufferPool constructor and the getPage() method.
- Implement the access methods, HeapPage and HeapFile and associated ID classes. A good portion of these files has already been written for you.
- Implement the operator SeqScan.

- At this point, you should be able to pass the ScanTest system test, which is the goal for this lab.

Section 3 below walks you through these implementation steps and the unit tests corresponding to each one in more detail.

2.6.1 Transactions, locking, and recovery

As you look through the interfaces we have provided you, you will see a number of references to locking, transactions, and recovery. You do not need to support these features in this lab. The test code we have provided generates a fake transaction ID that is passed into the operators of the query; you should pass this transaction ID into other operators and the buffer pool.

3 SimpleDB Architecture and Implementation Guide

SimpleDB consists of:

- Classes that represent fields, tuples, and tuple schemas;
- Classes that apply predicates and conditions to tuples;
- One or more access methods (e.g., heap files) that store relations on disk and provide a way to iterate through tuples of those relations;
- A collection of operator classes (e.g., select, join, insert, delete, etc.) that process tuples;
- A buffer pool that caches active tuples and pages in memory and handles concurrency control and transactions (neither of which you need to worry about for this lab); and,
- A catalog that stores information about available tables and their schemas.

SimpleDB does not include many things that you may think of as being a part of a "database." In particular, SimpleDB does not have:

- (In this lab), a SQL front end or parser that allows you to type queries directly into SimpleDB. Instead, queries are built up by chaining a set of operators together into a hand-built query plan. We will provide a simple parser for use in later labs.
- Views.
- Data types except integers and fixed length strings.
- (In this lab) Query optimizer.
- (In this lab) Indices.

In the rest of this Section, we describe each of the main components of SimpleDB that you will need to implement in this lab. You should use the exercises in this discussion to guide your implementation. This document is by no means a complete specification for SimpleDB; you will need to make decisions about how to design and implement various parts of the system. Note that for Lab 1 you do not need to implement any operators (e.g., select, join, project) except sequential scan. You will add support for additional operators in future labs.

3.1 The Database Class

The Database class provides access to a collection of static objects that are the global state of the database. In particular, this includes methods to access the catalog (the list of all the tables in the database), the buffer pool (the collection of database file pages that are currently resident in memory), and the log file. You will not need to worry about the log file in this lab. We have implemented the Database class for you. You should take a look at this file as you will need to access these objects.

3.2 Fields and Tuples

Tuples in SimpleDB are quite basic. They consist of a collection of `Field` objects, one per field in the `Tuple`. `Field` is an interface that different data types (e.g., integer, string) implement. `Tuple` objects are created by the underlying access methods (e.g., heap files, or B-trees), as described in the next section. Tuples also have a type (or schema), called a *tuple descriptor*, represented by a `TupleDesc` object. This object consists of a collection of `Type` objects, one per field in the tuple, each of which describes the type of the corresponding field.

Exercise 1: [2 points] Implement the skeleton methods in:

1. src/java/simpledb/TupleDesc.java
2. src/java/simpledb/Tuple.java

Unit Tests to Pass: At this point, your code should pass the unit tests `TupleTest` and `TupleDescTest`. Unit test `modifyRecordId()` should fail because you haven't implemented it yet.

3.3 Catalog

The catalog (class `Catalog` in SimpleDB) consists of a list of the tables and schemas of the tables that are currently in the database. You will need to support the ability to add a new table, as well as getting information about a particular table. Associated with each table is a `TupleDesc` object that allows operators to determine the types and number of fields in a table.

The global catalog is a single instance of `Catalog` that is allocated for the entire SimpleDB process. The global catalog can be retrieved via the method `Database.getCatalog()`, and the same goes for the global buffer pool (using `Database.getBufferPool()`).

Exercise 2: [2 points] Implement the skeleton methods in:

1. src/java/simpledb/Catalog.java

Unit Tests to Pass: At this point, your code should pass the unit tests in `CatalogTest`.

3.4 BufferPool

The buffer pool (class `BufferPool` in SimpleDB) is responsible for caching pages in memory that have been recently read from disk. All operators read and write pages from various files on disk through the buffer pool. It consists of a fixed number of pages, defined by the `numPages` parameter to the `BufferPool` constructor. For this lab, you only need to implement the constructor and the `BufferPool.getPage()` method used by the SeqScan operator. The BufferPool should store up to `numPages` pages. For this lab, if more than `numPages` requests are made for different pages, then instead of implementing an eviction policy, you may throw a DbException.

The `Database` class provides a static method, `Database.getBufferPool()`, that returns a reference to the single BufferPool instance for the entire SimpleDB process.

Exercise 3: [3 points] Implement the `getPage()` method in:

1. src/java/simpledb/BufferPool.java

Unit Tests to Pass: We have not provided unit tests for BufferPool. The functionality you implemented will be tested in the implementation of HeapFile below. You should use the `DbFile.readPage` method to access pages of a DbFile.

3.5 HeapFile Access

Access methods provide a way to read or write data from disk that is arranged in a specific way. Common access methods include heap files (unsorted files of tuples) and B-trees; for this assignment, you will only implement a heap file access method, and we have written some of the code for you.

A `HeapFile` object is arranged into a set of pages, each of which consists of a fixed number of bytes for storing tuples, (defined by the constant `BufferPool.DEFAULT_PAGE_SIZE`), including a header. In SimpleDB, there is one `HeapFile` object for each table in the database. Each page in a `HeapFile` is arranged as a set of slots, each of which can hold one tuple (tuples for a given table in SimpleDB are all of the same size). In addition to these slots, each page has a header that consists of a bitmap with one bit per tuple slot. If the bit corresponding to a particular tuple is 1, it indicates that the tuple is valid; if it is 0, the tuple is invalid (e.g., has been deleted or was never initialized.) Pages of `HeapFile` objects are of type `HeapPage` which implements the `Page` interface. Pages are stored in the buffer pool but are read and written by the `HeapFile` class.

SimpleDB stores heap files on disk in more or less the same format they are stored in memory. Each file consists of page data arranged consecutively on disk. Each page consists of one or more bytes representing the header, followed by the *page size* bytes of actual page content. Each tuple requires *tuple size* * 8 bits for its content and 1 bit for the header. Thus, the number of tuples that can fit in a single page is:

```
_tuples per page_ = floor((_page size_ * 8) / (_tuple size_ * 8 + 1))
```

Where *tuple size* is the size of a tuple in the page in bytes. The idea here is that each tuple requires one additional bit of storage in the header. We compute the number of bits in a page (by mulitplying page size by 8), and divide this quantity by the number of bits in a tuple (including this extra header bit) to get the number of tuples per page. The floor operation rounds down to the nearest integer number of tuples (we don't want to store partial tuples on a page!)

Once we know the number of tuples per page, the number of bytes required to store the header is simply:

```
headerBytes = ceiling(tupsPerPage/8)
```

The ceiling operation rounds up to the nearest integer number of bytes (we never store less than a full byte of header information.)

The low (least significant) bits of each byte represents the status of the slots that are earlier in the file. Hence, the lowest bit of the first byte represents whether or not the first slot in the page is in use. The second lowest bit of the first byte represents whether or not the second slot in the page is in use, and so on. Also, note that the high-order bits of the last byte may not correspond to a slot that is actually in the file, since the number of slots may not be a multiple of 8. Also note that all Java virtual machines are big-endian.

Exercise 4: [5 points] Implement the skeleton methods in:

1. src/java/simpledb/HeapPageId.java
2. src/java/simpledb/RecordID.java

3. src/java/simpledb/HeapPage.java

Although you will not use them directly in Lab 1, we ask you to implement `getNumEmptySlots()` and `isSlotUsed()` in HeapPage. These require pushing around bits in the page header. You may find it helpful to look at the other methods that have been provided in HeapPage or in `src/java/simpledb/HeapFileEncoder.java` to understand the layout of pages.

You will also need to implement an Iterator over the tuples in the page, which may involve an auxiliary class or data structure.

Unit Tests to Pass: At this point, your code should pass the unit tests in `HeapPageIdTest`, `RecordIDTest`, and `HeapPageReadTest`.

After you have implemented HeapPage, you will write methods for HeapFile in this lab to calculate the number of pages in a file and to read a page from the file. You will then be able to fetch tuples from a file stored on disk.

Exercise 5: [3 points] Implement the skeleton methods in:

1. src/java/simpledb/HeapFile.java

To read a page from disk, you will first need to calculate the correct offset in the file. Hint: you will need random access to the file in order to read and write pages at arbitrary offsets. You should not call BufferPool methods when reading a page from disk.

You will also need to implement the `HeapFile.iterator()` method, which should iterate through through the tuples of each page in the HeapFile. The iterator must use the `BufferPool.getPage()` method to access pages in the `HeapFile`. This method loads the page into the buffer pool and will eventually be used (in a later lab) to implement locking-based concurrency control and recovery. Do not load the entire table into memory on the open() call – this will cause an out of memory error for very large tables.

Unit Tests to Pass: At this point, your code should pass the unit tests in `HeapFileReadTest`.

3.6 Operators

Operators are responsible for the actual execution of the query plan. They implement the operations of the relational algebra. In SimpleDB, operators are iterator based; each operator implements the `DbIterator` interface.

Operators are connected together into a plan by passing lower-level operators into the constructors of higher-level operators, i.e., by 'chaining them together.' Special access method operators at the leaves of the plan are responsible for reading data from the disk (and hence do not have any operators below them).

At the top of the plan, the program interacting with SimpleDB simply calls `getNext` on the root operator; this operator then calls `getNext` on its children, and so on, until these leaf operators are called. They fetch tuples from disk and pass them up the tree (as return arguments to `getNext`); tuples propagate up the plan in this way until they are output at the root or combined or rejected by another operator in the plan.

For this lab, you will only need to implement one SimpleDB operator.

Exercise 6: [3 points] Implement the skeleton methods in:

1. src/java/simpledb/SeqScan.java

This operator sequentially scans all of the tuples from the pages of the table specified by the `tableid` in the constructor. This operator should access tuples through the `DbFile.iterator()` method.

Unit Tests to Pass: At this point, you should be able to complete the ScanTest system test. Good work!

You will fill in other operators in subsequent labs.

3.7 A simple query

The purpose of this section is to illustrate how these various components are connected together to process a simple query.

Suppose you have a data file, "some_data_file.txt", with the following content:

```
1,1,1,5
2,2,2,6
3,4,4,7
```

No matter what data you add to the file, make sure to **add an empty line at the end of the data**. You can convert this into a binary file that SimpleDB can query as follows:

```
java -jar dist/simpledb.jar convert some_data_file.txt 4
```

Here, the argument "4" tells convert that the input has 4 columns.

The following code implements a simple selection query over this file. This code is equivalent to the SQL statement

```
SELECT * FROM some_data_file
```

```
package simpledb;
import java.io.*;

public class test {

    public static void main(String[] argv) {

        // construct table schema
        Type types[] = new Type[]{ Type.INT_TYPE, Type.INT_TYPE, Type.INT_TYPE, Type.INT_TYPE };
        String names[] = new String[]{ "f0", "f1", "f2", "f3" };
        TupleDesc descriptor = new TupleDesc(types, names);

        // create the table, associate it with some_data_file.dat
        // and tell the catalog about the schema of this table.
        HeapFile table1 = new HeapFile(new File("some_data_file.dat"), descriptor);
        Database.getCatalog().addTable(table1, "test");

        // construct the query: we use a simple SeqScan, which spoonfeeds
        // tuples via its iterator.
        TransactionId tid = new TransactionId();
        SeqScan f = new SeqScan(tid, table1.getId());

        try {
            // and run it
            f.open();
            while (f.hasNext()) {
                Tuple tup = f.next();
                System.out.println(tup);
            }
            f.close();
            Database.getBufferPool().transactionComplete(tid);
        } catch (Exception e) {
```

```
            System.out.println ("Exception : " + e);
        }
    }

}
```

The table we create has four integer fields. To express this, we create a `TupleDesc` object and pass it an array of `Type` objects, and optionally an array of `String` field names. Once we have created this `TupleDesc`, we initialize a `HeapFile` object representing the table stored in `some_data_file.dat`. Once we have created the table, we add it to the catalog. If this were a database server that was already running, we would have this catalog information loaded. We need to load it explicitly to make this code self-contained.

Once we have finished initializing the database system, we create a query plan. Our plan consists only of the `SeqScan` operator that scans the tuples from disk. In general, these operators are instantiated with references to the appropriate table (in the case of `SeqScan`) or child operator (in the case of e.g. Filter). The test program then repeatedly calls `hasNext` and `next` on the `SeqScan` operator. As tuples are output from the `SeqScan`, they are printed out on the command line.

We **strongly recommend** you try this out as an end-to-end test that will help you get experience writing your own test programs for SimpleDB. You should create the file "test.java" in the src/simpledb directory with the code above, and place the `some_data_file.dat` file in the top level directory. Then run:

```
ant
java -classpath dist/simpledb.jar simpledb.test
```

Note that `ant` compiles `test.java` and generates a new jarfile that contains it.

4 Logistics

You must submit your code (see below) as well as a short (2 pages, maximum) writeup describing your approach. This writeup should:

- Describe any design decisions you made. These may be minimal for Lab 1.
- Discuss and justify any **changes you made to the API**.
- Describe any **missing or incomplete elements of your code**.
- Describe how long you spent on the lab, and whether there was anything you found particularly difficult or confusing.

4.1 Submission

You will package your entire project into a folder with the title: 'Lab1-groupID'. **Zip this folder** and submit via Moodle.

Only **one student per group** should submit and that student should always submit/resubmit to avoid multiple submissions per group.

In each folder, please include a readme file that includes a comma separated list of each member's full name.

We will not grade any submission that does not strictly follow the submission rules.

4.2 Collaboration

This lab is designed for a **group** of two persons. Larger groups are not allowed. Writing/exchange code across different groups is not allowed.

4.3 Grading

75% of your grade will be based on whether or not your code passes the system test suite we will run over it. These tests will be a superset of the tests we have provided. Before handing in your code, you should make sure it produces no errors (passes all of the tests) from all relevant ant test and ant systemtest.

Important: before testing, we will replace your build.xml and the entire content of the test directory with our version of these files. This means you cannot change the format of .dat files! You should also be careful changing our APIs. You should test that your code compiles the unmodified tests.

In other words, we will replace the files mentioned above, compile it, and then grade it. It will look roughly like this:

```
[replace build.xml and test]
...
$ ant test
$ ant systemtest
[additional tests]
```

If any of these commands fail, we'll have to fix things manually and this problem will affect your grade. Indeed, the additional 25% of your grade will be based on the quality of your writeup and our subjective evaluation of your code.

A Appendix

A.1 Java

Make sure you have the JDK installed on your machine. Linux is recommended (a VM is fine) to complete these assignments. It is possible to code in a Windows environment, however be aware that this may lead to complications that could be avoided by using Linux, since all the scripts that will be used are expecting a Linux system.

You may need to change the JDKPATH to the SDK path on your machine. This can be done by using the command `readlink -f $(which java)` in a terminal window. The output of the command should be similar to `/usr/lib/jvm/java-11-openjdk-amd64/bin/java` : the SDK path will in this case is then `/usr/lib/jvm/java-11-openjdk-amd64`.

A.2 Debug in Eclipse

Since SimpleDB is programmed in Java, it relies on the language's object-oriented features to implement the different parts of the code. It may be difficult to parse the flow of execution simply from looking at the code, so this section will go through one of the operations carried out by the DBMS when a class is run as an application.

The follow explanation assumes that you are using the debugger provided with Eclipse, however any other IDE may be used (generally, most if not all debuggers include the functions considered here).

By using a debugger, it will be possible to follow the thread of execution of the code and observe the content of variables of interest.

Text in `monospace` denotes either a function or a debugging command. You can hover the mouse pointer over the debugging icons to see the name of the command they'll execute (e.g. `Step into`, `Step over`).

- To run the application in debugging mode, select `Run` → `Debug`.
- Set breakpoints by double-clicking on the bar to the left of the line numbers.
- The content of variables can be shown by accessing the `Variables` panel in `Window`.
- Once a breakpoint is reached, it is possible to `Step into` or `Step over` the line.
- `Resume` the execution to let the code run until the next breakpoint, or until the execution is complete.
- It is possible to switch to the declaration of a function by pressing `CTRL + Left click` on a Windows/Linux system.

A.3 FAQ from previous labs

A1: Make sure Java and the JDK is installed on your system. Check here

Q2: The command `readlink -f $(which java)` returns the error

```
readlink: illegal option -- f
usage: readlink [-n] [file …]
```

A2: You probably are using MacOS. Check this. The path of the Java dir should be similar to `/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk`.

Q3: How can I integrate Eclipse with WSL 2?

A3: It's easier to run Eclipse from an Ubuntu Virtualbox. Check here

Q4: The command `readlink -f $(which java)` returns the error `readlink: missing operand`.

A4: Type `java -version` to make sure that Java is installed. If it's not, refer to Q2.